Project title: Community Networks Testbed for the Future Internet

Software system for the testbed

Deliverable number: D2.7

Version 1.0

This project has received funding from the European Union’s Seventh Programme for research, technological development and demonstration under grant agreement No 288535
**Project Acronym:** CONFINE  
**Project Full Title:** Community Networks Testbed for the Future Internet  
**Type of contract:** Integrated project (IP)  
**contract No:** 288535  
**Project URL:** [http://confine-project.eu](http://confine-project.eu)

<table>
<thead>
<tr>
<th>Editor:</th>
<th>Felix Freitag, UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable nature:</td>
<td>Report (R)</td>
</tr>
<tr>
<td>Dissemination level:</td>
<td>Public (PU)</td>
</tr>
<tr>
<td>Contractual Delivery Date:</td>
<td>September 30, 2015</td>
</tr>
<tr>
<td>Actual Delivery Date</td>
<td>September 24, 2015</td>
</tr>
<tr>
<td>Suggested Readers:</td>
<td>Project partners</td>
</tr>
<tr>
<td>Number of pages:</td>
<td>98</td>
</tr>
<tr>
<td>Keywords:</td>
<td>WP2, testbed design, developments, requirement analysis, experimental research, community networks, testbed</td>
</tr>
<tr>
<td>Authors:</td>
<td>Roger Baig (Guifi.net), Christoph Barz (FKIE), Christoph Fuchs (FKIE), Jonathan Kirchhoff (FKIE), Julia Niewiejska (FKIE), Bart Braem (iMinds), Emmanouil Dimogerontakis (UPC), Pau Escrich (Guifi.net), Felix Freitag (UPC), Esunly Medina (UPC), Aaron Kaplan (Funkfeuer), Ralf Schlatterbeck (Funkfeuer), Ivan Vilata (Pangea)</td>
</tr>
<tr>
<td>Peer review:</td>
<td>Leandro Navarro, Ester López (UPC)</td>
</tr>
</tbody>
</table>

**Abstract**

This document reports on the enhancements made in the system architecture, design and development for the CONFINE testbed software system (Community-Lab) done during the forth year of the project along with an overall summary of the final systems. It builds on the work reported in D2.5 during the third year of the project and previous deliverables D2.1, D2.2 and D2.3 of the first and second year of the project, respectively.

The software can be found in the project repository at [http://redmine.confine-project.eu](http://redmine.confine-project.eu).
5.1 Motivation for a Common Node Database ........................................... 39
5.2 Architecture of the NodeDB ................................................................. 40
5.3 Application Programming Interface Application Programming Interface (API) . 40
5.4 Node Database Applications ................................................................. 42
5.5 Dashboard ............................................................................................... 42
5.6 IP Address Reservation ........................................................................ 45
  5.6.1 Quota ............................................................................................... 45
  5.6.2 Freeing of Addresses Previously Allocated ................................. 45
  5.6.3 Access Control ................................................................................ 46
5.7 Spider ........................................................................................................ 46
5.8 Conclusions and Lessons Learned ......................................................... 47

6 Testing Framework .................................................................................... 48
  6.1 Introduction ............................................................................................ 48
  6.2 Architecture ........................................................................................... 48
  6.3 Tests ........................................................................................................ 49
  6.4 Jenkins interface .................................................................................... 50
    6.4.1 Overview ......................................................................................... 50
    6.4.2 Test matrix ...................................................................................... 51
    6.4.3 Test results ..................................................................................... 52
    6.4.4 Test coverage .................................................................................. 53
  6.5 File system structure ............................................................................. 53
  6.6 Statistical Evaluation of Impact ............................................................. 54
    6.6.1 Bug reports ..................................................................................... 54
    6.6.2 Resolved bugs ............................................................................... 55
  6.7 Test Running and Requirements .......................................................... 55
    6.7.1 Requirements ............................................................................... 55
    6.7.2 How to ............................................................................................ 56
    6.7.3 Run tests manually ....................................................................... 57
    6.7.4 Run an installation test ................................................................. 58
    6.7.5 Control which tests to run ............................................................ 58
    6.7.6 Build your own VCT container .................................................... 59
  6.8 Overview of testing resources ............................................................... 59
  6.9 Conclusion .............................................................................................. 59

7 VCT/ns-3 Integration .................................................................................. 60
  7.1 ns-3 installation .................................................................................... 60
  7.2 Connectivity design .............................................................................. 61
  7.3 Usability adjustments .......................................................................... 62
  7.4 Conclusions ......................................................................................... 66

8 Dynamic Link Exchange Protocol (Dynamic Link Exchange Protocol (DLEP)) 67
  8.1 Implementation ...................................................................................... 67
  8.2 Integration ............................................................................................. 67
  8.3 Conclusion ............................................................................................. 68

9 Testbed monitoring system ........................................................................ 70
  9.1 Motivation .............................................................................................. 70
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The network architecture of a Community-Lab testbed.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>The node/functional architecture of a Community-Lab testbed.</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Node architecture in the “Confined” milestone</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Data model in the “Confined” milestone</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>WiBed design overview.</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Node firmware filesystem in the IDLE status.</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Node firmware filesystem in the RUNNING status.</td>
<td>25</td>
</tr>
<tr>
<td>4.4</td>
<td>Simplified functional diagram for a node</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>Node finite-state machine.</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>Main page of WiBed web interface.</td>
<td>28</td>
</tr>
<tr>
<td>4.7</td>
<td>WiBed web interface - Admin Tab.</td>
<td>29</td>
</tr>
<tr>
<td>4.8</td>
<td>WiBed web interface - Nodes Tab.</td>
<td>30</td>
</tr>
<tr>
<td>4.9</td>
<td>WiBed web interface - Node Page.</td>
<td>31</td>
</tr>
<tr>
<td>4.10</td>
<td>WiBed web interface - Errors Tab.</td>
<td>32</td>
</tr>
<tr>
<td>4.11</td>
<td>WiBed web interface - Topology Tab.</td>
<td>33</td>
</tr>
<tr>
<td>4.12</td>
<td>WiBed web interface - Experiments Tab.</td>
<td>34</td>
</tr>
<tr>
<td>4.13</td>
<td>WiBed web interface - Add Experiment Page.</td>
<td>34</td>
</tr>
<tr>
<td>4.14</td>
<td>WiBed web interface - Show Experiment Page.</td>
<td>35</td>
</tr>
<tr>
<td>4.15</td>
<td>WiBed web interface - Results Tab.</td>
<td>35</td>
</tr>
<tr>
<td>4.16</td>
<td>WiBed web interface - Firmwares Tab.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of the overall architecture of CNDB</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Object Model of Node Database</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Dashboard example</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Dashboard example small device</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>CONFINE automated testing architecture</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Jenkins’ job overview</td>
<td>51</td>
</tr>
<tr>
<td>6.3</td>
<td>Jenkins’ test matrix</td>
<td>52</td>
</tr>
<tr>
<td>6.4</td>
<td>Jenkins’ result trends</td>
<td>52</td>
</tr>
<tr>
<td>6.5</td>
<td>Jenkins’ test results</td>
<td>52</td>
</tr>
<tr>
<td>6.6</td>
<td>Jenkins’ test coverage</td>
<td>53</td>
</tr>
<tr>
<td>7.1</td>
<td>Example of connectivity between two slivers over Network Simulator 3 (ns-3) inside the VCT Container.</td>
<td>61</td>
</tr>
<tr>
<td>8.1</td>
<td>DLEP proxy in Confine architecture</td>
<td>68</td>
</tr>
<tr>
<td>9.1</td>
<td>Receiver-Pull Model followed by the Community-Lab Monitor</td>
<td>71</td>
</tr>
<tr>
<td>9.2</td>
<td>Components of the Monitoring Client located in the Research Device</td>
<td>72</td>
</tr>
</tbody>
</table>
9.3 Sequence number management .................................................. 73
9.4 Timestamp management ......................................................... 74
9.5 Components of the Monitoring Server located in the Community-Lab Controller . . . 75
9.6 Home screen of the Monitoring system for Confine .......................... 75
9.7 Generation of the “most recent” documents by the monitoring server ......................... 77
9.8 How documents are stored in the database using a document identifier ....................... 77
9.9 Monitored metrics in the Research Device .................................... 79
9.10 Average CPU usage ............................................................... 80
9.11 Sliver usage ..................................................................... 81
9.12 Treemap representation ......................................................... 81
9.13 Synthesized metrics monitored by the server .................................. 82
9.14 Network topology map ............................................................ 82
9.15 Example of selection of RDs connected to the same CD ...................... 83
9.16 Traceroute pluggable daemon .................................................... 83
9.17 Traceroute metrics monitored by the server ................................... 84
9.18 Example of the trace information monitored by the RDs ....................... 84
9.19 Example connectivity of a particular RD to other RDs in the network. Link length is characterised by the latency between the RDs ........................ 85
9.20 Global link quality information across all RDs depicted as a matrix. The intensity of "red" provides a visual information of the latency. Darker red indicates higher latency . . . 86
9.21 Example of the operation of views in CouchBase .............................. 87
9.22 Example of the get_node-id view ............................................. 89
9.23 Example of the get_node-cpu_usage_percentage ............................ 90
9.24 Example of the view get_all_nodes-cpu_usage_statistics .................. 90
9.25 Example of the view get_all_nodes-cpu_usage_statistics using group level 1 .... 91
9.26 Example of the view get_all_nodes-cpu_usage_statistics using group level 5 .... 91
9.27 Example of the view output for set metrics .................................. 92
List of Tables

4.1 Node states ......................................................... 26
6.1 The relationship between the tests and bug reports. ......................... 54
6.2 The relationship between the tests and bug reports pre and post the introduction of the test framework. ......................................................... 55
6.3 The relationship between the resolved bug rates and the open bug rate. .......... 55
6.4 Testing resources overview ............................................ 59
9.1 Overview of monitoring resources ........................................ 93
1 Introduction

This document reports on the final version of the Community-Lab architecture, system software, software tools and services that form the extended Community-Lab testbed environment for experimentation.

The main results are:

- **Overall architecture**: Presents an integrated view of the architecture of the testbed facilities, from a network perspective with the interconnection of several community networks, and from a component (and functional) perspective, presenting the organization of the Community Network testbed, the WiBed Campus testbed and the VCT Virtual testbed.

- **Core testbed components**: mainly the node software, controller software, and Virtual Confine Testbed (VCT) container.

- **WiBed: The Campus Testbed**: The WiBed node and the WiBed controller.

- **Common NodeDB**: The common node database, a management information database for wireless community networks.

- **Testing framework**: based on the Jenkins platform. Developed and used to perform automated tests and validation of many software components and interfaces.

- **Integration of VCT and the ns-3 simulator**: An enhancement to include in the virtual testbed VCT support for realistic network and link layer conditions using the ns-3 network simulator.

- **Dynamic Link Exchange Protocol (DLEP)**: a standard local connection between a bridging radio and a router to allow the router to learn about the known link-layer and physical layer data. The implementation is aligned to the standardisation work at the IETF and is integrated with Community-Lab research devices.

- **Confine monitoring system**: A distributed system designed to monitor activity of Research Devices (RD) in the Community-Lab testbed, collecting and presenting diverse, precise and historic metrics of performance.

All these contributions are deployed and in production use as part of the Community-Lab testbeds, by the community networks involved, and as tools used by experimenters in the area of community networking from academia, industry, community network participants and software developers.

1.1 Contents of the deliverable

This deliverable is the result of the software development work in the four years of the project and builds upon deliverables D2.5, reporting system enhancements in year 3, D2.3, reporting system enhancements in year 2, and deliverables D2.1 and D2.2 of the first year of the project. The software can be found in the project repository\(^1\).

The description of work document for CONFINE mentions this deliverable as instrumental to reflect progress in the following items.

\(^1\)http://redmine.confine-project.eu
1.2 Relationship to other project deliverables

In Indicators or advance over the state of the art:

“That work will be performed and revised over the 4 yearly iterations of the testbed in task T2.2. Revisions to the software will be released on D2.3 (M24), D2.5 (M36) and D2.7 (M48).”

“Software components that implement proposed algorithms and integrate them into the management frameworks are defined in T2.2 and delivered in D2.3 (M24), D2.5 (M36) and D2.7 (M48).”

“Software tools for implementing self-management actions for the automation of the testbed, dealing with the interrelated problems of distributed (global) allocation of channels, IP addresses, IP ranges, routes, routing domains. These tools will be integrated in the enhancements to the management tools and services and the embedded system software developed in T2.2 and delivered in D2.3 (M24), D2.5 (M36) and D2.7 (M48).”

As part of T2.2, this deliverable is described as:

“The enhancements of tools and services, and the update of embedded node system will be reported in D2.3 (M24) (software and documentation), D2.5 (M36) (software and document), D2.7 (M48), describing in detail the problem addressed and the developed solution. D2.7 (M48) will be the final software system and a complete report of the final system.”

1.2 Relationship to other project deliverables

D2.5 System enhancements (Year 3) – M36: This deliverable reports the consolidated CONFINE testbed software system achieved during the third year of the project. D2.5 updates D2.3. It describes extension and consolidations of some of the components, while other components developed in the second year were not further pursued for integration into the production testbed.

D2.3 System enhancements (Year 2) – M24: A deliverable that reports on the architecture, design and developments of the CONFINE testbed software system done during the second year of the project. Some components were in experimental stage.

D2.8 Implementation of federation mechanisms for community networks – M36: Describes the federation mechanisms implemented for the CONFINE testbed which are used to interconnect the diverse community networks involved in the project and other testbed facilities. The final version of the testbed maintains being federated as to the description in D2.8.

D4.4 Experimental research on testbed for community networks (year 4) – M48: D4.4 reports on experimentally driven research in the fourth year to support the development of the CONFINE testbed software system. The interaction between WP2 and WP4 remained bidirectional. The final software system of the testbed took into account research on the state-of-the-art. D4.4 therefore contains research work that contributed to the final CONFINE testbed software system.

D5.6 Dissemination, training, standardization activities in year 4 – M48: This deliverable reports on the interactions which the CONFINE project had with different stakeholders in many kinds of events. The CONFINE project was communicated to third parties and CONFINE also received valuable external feedback. Among others the aspect of continuity was important for the direction of the work in the CONFINE testbed software system took in the forth year of the project.

D5.7 Socio-technical-economic-legal evaluation and sustainability model – M48: This deliverable reports on the sustainability model. The findings were relevant for to the direction of the final CONFINE testbed software system.

D5.9 Exploitation Plan – M48: The overall and individual exploitation plans were taken into account in the final software system of the CONFINE testbed.
2 Overall Architecture

This chapter is a revision and extension of the overall architecture in D2.1 \[1\] with additional details describing the overall architecture of the infrastructure available for experimentation from a network perspective, and from a component (and functional) perspective.

2.1 Network perspective

Figure 2.1 shows two community networks (CNs) with several community nodes connected to them. These community nodes are integral parts of a Community Network (CN) and form the actual network infrastructure by relaying other nodes’ traffic while respecting the rules and peering agreements of the CN. The community nodes and the network infrastructure itself are managed by community members (depicted as node administrator the person on the right).

In this context, a Community-Lab testbed consists of at least one testbed server (or controller) and a set of testbed nodes spread among (and connected to, through wired or wireless interfaces) the existing community nodes of one or several CNs. The testbed nodes (as an aggregate) are orchestrated by the testbed servers.

A Community-Lab testbed server is a normal computer which must be directly reachable from inside a CN using this network’s Internet Protocol (IP) addresses, and usually (but not necessarily) also from the Internet using public IP addresses. It is managed by testbed administrators (the person in the centre of Figure 2.1) who need not be CN administrators.

A Community-Lab testbed node is a device that plugs behind existing community nodes and implements access control, resource isolation and management capabilities with the objective to grant...
Figure 2.2: The node/functional architecture of a Community-Lab testbed.

external researchers/users a confined access to the node’s processing and network resources, while enforcing respect of the CN rules. If the testbed node belongs to a community member, he or she must also adhere to the testbed policies and conditions. This decouples testbed management from infrastructure ownership and management.

Finally, testbed researchers (the person on the left of Figure 2.1) use testbed servers to configure their experiments and run them in testbed nodes, possibly with direct interaction.

As of September 2015, the status of implementation of the above architecture is as follows:

- RDs are deployed in 6 community networks: guifi.net (ES), Ninux.org (IT), Funkfeuer.at (AT), AWMN.net (GR), Sarantaporo.gr (GR), and WirelessBelgië (BE). There are 210 testbed nodes (RD) registered in the testbed.
- There is one main testbed server (the Community-Lab.net controller) operated by Pangea and hosted by UPC in the guifi.net network, one more for testing run by iMinds, and another for local experiments run by Fraunhofer FKIE. Additionally there is one Federation server operated by UPC in the academic network and one Monitoring server, operated by UPC and connected both to the academic network and the guifi.net network.
- The network interconnection is now using the Géant academic Virtual Local Area Network (VLAN)/tunneling infrastructure supported by the Spanish National Research and Education Network (NREN) (RedIRIS) and additional IP over IP tunnels to reach a couple of CN or as a backup.
2. Overall Architecture

2.2. Component and Functional perspective

- A thin decentralized overlay network (with a set of gateway nodes) provides a protected, transparent and bidirectional IPv6 virtual network that offers seamless connectivity (bidirectional and end-to-end, despite several CN can use IPv4 and others IPv6, and some of them can use overlapping and therefore incompatible addressing schemes) across all RD and servers (and computers from experimenters participating in the experiment) that also protects the testbed from external attacks.

2.2 Component and Functional perspective

Figure 2.2 shows the functional architecture of the three environments for experimentation in Community-Lab (as columns in the figure): the CN testbed deployed in several CNs, the virtual testbed (can be instantiated within a single computer) and the campus testbed (can be deployed in specific areas). These experimentation environments can be classified in three parts (as rows in the figure): servers, testbed nodes and routers.

- The main testbed in Community-Lab is the Community Network testbed with testbed nodes (Research Devices) deployed as hosts embedded in several production CNs, depicted in the left column. It has three main servers/functions: a) the Federation service (Slice-based Federation Architecture (SFA)) that interconnects with the Fed4FIRE federation, b) the Controller service that allows the experimenter to define slices and select nodes (RD) to deploy sliver templates, informs each RD about its desired state (slivers requested for a given RD), and collects and aggregates sliver status information, and c) the Monitor that collects, aggregates and presents monitoring information from each RD (with a local monitoring daemon in each RD).

Each testbed node (Research Device (RD)) has three types of interfaces: a ssh interface to access the host by the node owner and for each sliver by each experimenter allowed in the slice/sliver configuration defined in the controller, the monitoring interface, and the node management (resource control) interface. Each RD is installed with a custom system image that is prepared in the controller. Each sliver is instantiated from a container image. The RD has multiple virtual network interfaces: for the host, for the slivers and for management.

RDS are connected to the underlying CN that can be seen as an IP network of interconnected routers. These routers can have internal network interfaces (wired or wireless) and also have network attached radios (using the DLEP specification developed by the project).

Each CN has its own node database that can provide detailed information about each router in the CN, such as interfaces, traffic, location, owner, etc. Each CN has typically its own database, with one NodeDB developed in this project with the goal of providing a common scheme.

- The Campus testbed (also known as WiBed) is both a specific deployment at the UPC campus and a software package that can also be deployed in other locations, as it has been done during the BattleMesh-v7 2014 in Leipzig (DE) and BattleMesh-v8 2015 in Maribor (SL). It is depicted as the right column in Figure 2.2.

The WiBed testbed nodes can run experimental firmware images that are downloaded from a control server, the WiBed controller, with a web and API control interface to control nodes and experiments. This testbed has a specific architecture to allow radio and router level experiments in a separate network for low-level and potentially network disturbing and radio interfering experiments that may not be compatible with a production CN. The most typical experiment has been the testing and performance characterisation of mesh routing protocols.

Around 40 router-like devices are deployed at different offices in several buildings in the UPC
North Campus (Barcelona). These router devices have three radios. A few of them also include an Ethernet connection for control.

- The **Virtual testbed** (also known as VCT) is a software package that allows to deploy a complete testbed in a single computer (or a cluster) using virtual machines and a virtual network, that can also simulate virtual wireless links combined with the ns-3 network simulator. It is depicted as the central column in Figure 2.2. A VCT instance of a virtual testbed should contain one controller, a set of virtual RDs and a set of virtual routers, and a virtual network that can also simulate radio links. The main purpose of VCT is to provide a controlled environment to allow debugging experiments before (or in parallel from) being run in the main CN testbed. For that reason, the software is nearly the same as in the main testbed and therefore an experiment can be easily moved between the virtual and the real testbeds.
3 Core testbed components

3.1 Introduction

The CONFINE Node software [2] and CONFINE Controller software [3] have received a great amount of deep changes during the last year. Besides the usual fixes to several problems arisen during the usage of Community-Lab [4] and other CONFINE-based testbeds, the main changes have been oriented towards the stabilization of the CONFINE architecture, and the Node and Controller implementations themselves, and the maintainability and ease of adoption of the software by future users. To this point, new utilities [5] and tests [6] have been developed to complement the usage and development of the software. As usual, all the aforementioned software is available with a full history of changes and issues in the CONFINE Redmine server [7].

From July 15th 2014 to the present day (August 2015), around 150 new issues have been created in Redmine’s issue tracker, with less than 50 remaining open, some of them being long-term feature requests, and less than 10 of them having a high priority. This is an indication of the high effort put into the stabilization of the testbed and its reference implementation provided by the CONFINE Project. Issues related with the topics discussed in this chapter will be indicated as #NUMBER and linked in section 3.4.

The resulting CONFINE testbed architecture, which is implemented by the new stable releases of the Node software (Master 20150715-1002), Controller software (1.0.1) and VCT container [8] (201507291401), comprises the “Confined” milestone [9], planned for the stability and maintainability objectives mentioned above. The node architecture offered by this release can be seen in fig. 3.1, and the associated object/data model in fig. 3.2. This model is reflected by versions v1 of the Registry and Node REST APIs [10], which along the Controller API constitute the programmable interface to CONFINE testbeds.

3.2 Architectural updates

The main changes to the CONFINE architecture from the previous ”Bare bones” milestone [11] are:

- Support for per-testbed and per-node resources and their allocation for slices and slivers, respectively (#46, #602). Maximum and default requests can be set for each resource in the testbed or node (by superusers and node administrators, respectively). Explicit requests of disk space and memory are allowed for slivers, as well as implicit requests of public IPv4 and IPv6 addresses via sliver interfaces. Implicit requests of VLAN tags are allowed for slices via the activation of isolated interfaces.

- Separation of Registry and Controller APIs so that static configuration is stored in the Registry, and other Controller software-dependent operations are provided by the Controller API (#236). For instance, uploading files is now done via the Controller API, since the Registry has no concept of stored files, only URIs (#238).

- Unification of testbed server and gateways so that multiple servers can be defined (#236). The main difference of servers with nodes and hosts is that the former are maintained by testbed
Operators instead of normal users, and they can publish several API endpoints (like Registry and Controller).

- **Configurable API endpoints** for Node, Registry and Controller APIs in testbed nodes and servers (#245). Instead of the previous implicit API base URIs, this sticks to REST’s “Hypermedia as the Engine of Application State” (HATEOAS), and allows for flexible delegation on per-island caching proxies, front ends for groups of nodes, etc. Since the certificate is now also indicated for each API endpoint, there is no longer need for a centralized Certificate Authority run by testbed operators.

- Definition of **sliver defaults** in the slice which can be overridden by slivers and include resource requests, sliver template and data, and set state (#234). Along with the slice’s own set state, this allows sophisticated combinations like all-slivers-running-but-some or all-slivers-stopped-but-some.

- Make **network backend configuration** (tinc or native) independent from management network configuration in testbed hosts, nodes and servers (#157). This allows the back end to be reused by other features like a hypothetical VPN! (VPN!) to access the Internet from certain community networks.

- Unification of tinc server and client into **tinc host** to resemble the underlying mesh model (#157). Any such host (i.e. a testbed host, node or server) can act as a gateway to the testbed’s management network if trusted by the host connecting to it. This does away with the need of

---

**Figure 3.1:** Node architecture in the “Confined” milestone
dedicated gateways and enables more resilient overlay setups in islands without intervention from testbed superusers.

- Relate hosts and nodes directly to islands (#264), as well as server API endpoints and tinc addresses (#236, #245, #157). This eases the client API code needed to locate hosts that offer a specific API or service in an island.

Changes that simplify or enhance node operation are:

- Merge sliver overlay and data into sliver data (#200). This makes the concept more generic and relies its interpretation to the particular template type (e.g. as a configuration file for a
3.3 Implementation updates

Since the implementation of the architectural changes mentioned above reflects on both Node and Controller software, those changes and their related issues will not be mentioned again here. Ordinary bug fixes will not be covered either to avoid filling the chapter with many technical links (simply refer to the comment on issue counts in the introduction to get an idea on the bug fixing activity). Only the main fixes and implementation achievements will be summarized in this section. Also, many of these changes include tests, some of them covered in their own issues which will not be included here either for brevity.

3.3.1 General

On the whole, changes resulting from architectural updates have been implemented with backwards compatibility in mind. Thus, although recent Node firmwares are only compatible with similarly new Controller versions that implement parts of the stable CONFINE architecture, Controller versions prior to 1.0 still provide backwards compatibility members in Registry API resources for older firmwares. Thus, the upgrade plan for CONFINE testbeds is to run the latest pre-1.0 version of the Controller until nodes have been upgraded to the latest stable Master version, and then upgrade the Controller to 1.0, which does not include backwards compatibility features (#620).
A big development and testing effort has been put in upgrading the Node software to **OpenWrt 14.07 “Barrier Breaker”**, the current stable OpenWrt release, and supporting it in the Controller. Adopting this release increases the chances of adoption of CONFINE software by OpenWrt users and developers, as well as improving hardware support (among others).

With the new stable releases of Node and Controller software, a new VCT container [8] (version 201507291401) has been packaged that includes them both, along with other CONFINE tools, in a Debian Jessie base system that can be run under LXC [12] to get a complete virtual CONFINE testbed with nodes and a Controller, for learning and testing. Initial work has also been done to create a VCT Docker container [13]. We expect to leverage the software used to create these containers in order to prepare plain Controller containers in the future.

Node administrators now have a greater control over who can access their nodes as root. They have the possibility of setting a closed set of authorized keys, having the node poll the registry for new administrators in the group, or even accepting additional keys for centralized remote maintenance. Also, for an easier administration of the community devices where research devices (i.e. nodes) use to be attached, some implementation limitations on the arrangement of sliver interfaces have been coordinated between Node and Controller software (#633).

Finally, several new scripts have been added to the utilities repository. The REST API client library included there has also been updated to the stable API specification. Some integration code for creating KVM [14] nodes under the Cloudy distribution [15] can also be found there, along with the software used to package the LXC-based VCT container as mentioned above.

### 3.3.2 Node

**Regarding installation and upgrades:**

- Many issues related with node firmware upgrades have been fixed (#552, #634, #360, #228, #573, #635 and others). Now node upgrades [16] can be performed preserving its configuration and deployed slivers’ data.
- A reorganization of the boot process has been done which results in fewer reboots and faster installation and upgrades.
- The build process now supports images for four platforms (#483, #680): generic (386-compatible), i586 (Pentium-compatible with 4 GiB high memory support), i686 (Pentium Pro-compatible with SMP and 64 GiB PAE support) and Atom (i686 with specific optimizations). I586 and i686 images are distributed by the Project in the CONFINE node image repository [17].

**Regarding configuration and management:**

- The node’s local and direct interfaces can be VLAN-tagged. This allows hybrid nodes using DLEP to communicate with external network interfaces. On slivers with isolated interfaces, this implies VLAN stacking (802.1ad or QinQ, which requires Barrier Breaker).
- Node administrators can permanently customize the location of the firmware used in remote upgrades (#653, #654), e.g. for choosing a customized image for some nodes, or a closer image repository.
- The on-demand CPU governor has been enabled by default to save power on idle nodes (#59).
- Community-Lab specific customizations have been moved to a separate package (#655).

**Regarding testbed functionality:**

Deliverable D2.7
3.3 Implementation updates

- A long-standing bug in HTTP redirects of the Node API that caused bad status line errors in API clients has been fixed (#299).
- A new script for dumping a sliver overlay can be used to easily create a sliver data file e.g. from a testing sliver running in VCT.
- Open vSwitch packages have been updated to version 2.3 to support richer OpenFlow and software-defined networking (SDN) experiments.
- Debug interfaces have been implemented (#90) to allow reaching nodes and slivers in a testing setup on a single network link.
- Pending features from the CONFINE REST API specification [10] have been implemented (#171).

3.3.3 Controller

Regarding installation and maintenance:

- Some scalability issues with the growing size of the database have been fixed (#475, #448).
- Compatibility with the latest Debian Jessie (#685) and its Apache 2.4 packages (#684) has been added.
- The standard Controller installation now includes an NTP server which can be used by nodes over the management network to keep their clocks synchronized (#404).

Regarding testbed functionality:

- Pending features from the CONFINE REST API specification [10] have been implemented like filtering (#403), pagination (#409) and request validation (#536).
- Several functionalities have been added to the Controller API like sliver data uploading (#45), node state reporting (#460), firmware configuration (#470), base node image uploading (#434), and GIS data reporting (#382).
- Connections against the Node API are validated if the registry specifies a certificate for it (#585). This revealed an obscure issue with certificates created in the firmware generator, which has been fixed (#625).

Usability enhancements:

- The Controller makes a **broader use of notifications** towards testbed operators (#449) and node administrators (#523, #587).
- An interesting use of notifications is part of the new **sliver’s journal application**, which reports a history of sliver usage for the nodes of a group. This may improve the awareness of community network members about the utility of their nodes to researchers.
- A new **customizable dashboard** with information on the user’s hosts has been added (#47).
- **Better on-line documentation** is provided on firmware installation instructions (#674) and unavailable sliver interfaces (#216).
- Better access to the configuration of tinc addresses (#603).
- A new CONFINE administrator’s guide [18] has been written that mostly covers Controller usage for testbed operators, and which complements the already existing CONFINE user’s guide [19].

Ongoing updates:
3. Core testbed components

3.4 Issues

Links to the issues mentioned above, sorted in ascending numerical order:

- #45: Upload exp_data for slivers and slices:
  https://redmine.confine-project.eu/issues/45
- #46: enhancing specification of slice/sliver resource limits (eg disk space) via RestApi:
  https://redmine.confine-project.eu/issues/46
- #47: Usability issues:
  https://redmine.confine-project.eu/issues/47
- #59: Enable the ondemand CPU governor in RDs:
  https://redmine.confine-project.eu/issues/59
- #90: Debug interfaces in slivers are ignored:
  https://redmine.confine-project.eu/issues/90
- #157: Rearrange tinc and management network attributes:
  https://redmine.confine-project.eu/issues/157
- #171: Echo ”/rel/server/*” links from the server:
  https://redmine.confine-project.eu/issues/171
- #200: Allow user-provided overlay for sliver:
  https://redmine.confine-project.eu/issues/200
- #216: Provide some help on unavailable types of sliver ifaces:
  https://redmine.confine-project.eu/issues/216
- #228: confine.remote-upgrade preserve files that cause errors:
  https://redmine.confine-project.eu/issues/228
- #234: Factor sliver defaults out of Slice class:
  https://redmine.confine-project.eu/issues/234
- #236: Multi-server support:
  https://redmine.confine-project.eu/issues/236
- #237: Report all node addresses:
  https://redmine.confine-project.eu/issues/237
- #238: Move functions to controller API:
  https://redmine.confine-project.eu/issues/238
- #239: Remove firmware configuration cruft from data model:
  https://redmine.confine-project.eu/issues/239
3.4. Issues

- #245: Configurable API links in REST API:
  https://redmine.confine-project.eu/issues/245
- #264: Move "TincClient.island" to "Node.island" and "Host.island":
  https://redmine.confine-project.eu/issues/264
- #276: Use application/json with parameters as media-type rather than vnd specific:
  https://redmine.confine-project.eu/issues/276
- #299: Node software very sensitive to timeouts:
  https://redmine.confine-project.eu/issues/299
- #360: Run confine.disk-parted automatically on boot:
  https://redmine.confine-project.eu/issues/360
- #382: Provide REST API for GIS:
  https://redmine.confine-project.eu/issues/382
- #403: converting CONFINE’s filtering spec to Django’s:
  https://redmine.confine-project.eu/issues/403
- #404: Add time synchronisation to RDs:
  https://redmine.confine-project.eu/issues/404
- #409: REST API pagination support:
  https://redmine.confine-project.eu/issues/409
- #434: Support uploading new base image over API:
  https://redmine.confine-project.eu/issues/434
- #448: Pings objects not deleted on cascade:
  https://redmine.confine-project.eu/issues/448
- #449: Notify other operators when user is activated:
  https://redmine.confine-project.eu/issues/449
- #450: Include management address in sliver description:
  https://redmine.confine-project.eu/issues/450
- #460: Expose node state in controller API:
  https://redmine.confine-project.eu/issues/460
- #470: Add firmware plugin functionality to REST API:
  https://redmine.confine-project.eu/issues/470
- #475: Not possible to delete slices via admin interface:
  https://redmine.confine-project.eu/issues/475
- #479: Add ”id” member to reference objects in API:
  https://redmine.confine-project.eu/issues/479
- #483: Support for more than 4GB RAM on 32bit architectures:
  https://redmine.confine-project.eu/issues/483
- #510: Add name attribute to hosts:
  https://redmine.confine-project.eu/issues/510
- #523: Show sliver history for node?:
  https://redmine.confine-project.eu/issues/523
- #536: Implement Unprocessable Entity Error:
  https://redmine.confine-project.eu/issues/536
3. Core testbed components

3.4. Issues

- #552: Firmware upgrades failing:
  https://redmine.confine-project.eu/issues/552
- #573: confine.remote-upgrade:
  https://redmine.confine-project.eu/issues/573
- #582: Django 1.7 compat:
  https://redmine.confine-project.eu/issues/582
- #585: Check node API certificate if using HTTPS:
  https://redmine.confine-project.eu/issues/585
- #587: Notify when node is down for too long:
  https://redmine.confine-project.eu/issues/587
- #595: Report actual node state in node API:
  https://redmine.confine-project.eu/issues/595
- #602: Memory resource limits in slivers:
  https://redmine.confine-project.eu/issues/602
- #603: Simplify configuration of tinc addresses:
  https://redmine.confine-project.eu/issues/603
- #620: Drop backwards compatibility code:
  https://redmine.confine-project.eu/issues/620
- #625: uhttpd(2) problem with python generated certificate:
  https://redmine.confine-project.eu/issues/625
- #633: no sliver reachability with non-persistent DHCP leases:
  https://redmine.confine-project.eu/issues/633
- #634: Node upgrade results in endless loop:
  https://redmine.confine-project.eu/issues/634
- #635: Warn or stop when upgrading nodes in use:
  https://redmine.confine-project.eu/issues/635
- #645: Build node firmware using OpenWRT ImageBuilder:
  https://redmine.confine-project.eu/issues/645
- #648: report hardware characteristics of nodes via node RestAPI:
  https://redmine.confine-project.eu/issues/648
- #653: Configurable image URI for remote upgrade:
  https://redmine.confine-project.eu/issues/653
- #654: Allow Research nodes to download the latest firmware over the management network:
  https://redmine.confine-project.eu/issues/654
- #655: Missing customizations and the confine-community-lab package:
  https://redmine.confine-project.eu/issues/655
- #665: Relax SAFE to not undeploy slivers:
  https://redmine.confine-project.eu/issues/665
- #674: Update installation instructions:
  https://redmine.confine-project.eu/issues/674
- #677: Developer’s guide of controller:
  https://redmine.confine-project.eu/issues/677
3.4. Issues

3. Core testbed components

- #680: Enable PAE by default to support big memory:
  https://redmine.confine-project.eu/issues/680
- #684: Debian Apache 2.4 compatibility:
  https://redmine.confine-project.eu/issues/684
- #685: Firmware generation fails with Debian Jessie’s file:
  https://redmine.confine-project.eu/issues/685
4 WiBed: The Campus Testbed

WiBed is a Free and Open Source Software (FOSS) software platform for providing commodity wireless testbeds. It is deployed as a CONFINE testbed facilitating experiments that were not possible with Community-Lab, as we describe here. WiBed is based on the OpenWRT Linux made to run on commodity IEEE802.11 WiFi routers and it has been designed to support realistic low layer network experiments (according to the OSI model).

The WiBed architecture has been conceived to diminish the hardware restrictions: the capability of running a GNU/Linux system and having two ath9k supported Wireless Network Interface Cards (WNICs) are the minimum conditions set by design. Currently these conditions are broadly fulfilled by many of the Commercial off-the-shelf (COTS) wireless routers available in the retail market for less than 100€, allowing the deployment of WiBed-like testbed of tenths of nodes for a few thousand Euro. Thanks to a minimised management system, WiBed allows experiments from link-layer to application-layer, focused though on L1-L3 experiments. Community-Lab however is limited regarding conducting L1-L3 experiments, due to potential service interruptions in its environment that these experiments may produce. These disturbances are mitigated from running the experiments at night periods outside normal working or study hours. With regards to this feature, WiBed extends Community-Lab by enabling L1-L3 experiments.

Currently, a WiBed testbed of 40 nodes from the planned 50 nodes has been deployed over two buildings of Universitat Politècnica de Catalunya (UPC) Campus Nord, Barcelona. The resulting testbed is available to the researchers as part of the project. More details about the deployment can be found in D3.3. In addition, WiBed has been successfully used as basis platform for the Wireless Battle of the Mesh 2014\(^1\), where several routing experiments were executed. More details about the experiments performed in WiBed can be found in D4.3.

More information about this work can be found the following articles and documents:

- Wibed, a platform for commodity wireless testbeds(2014) [22]
- UPC CN-A testbed mesh network deployment, monitoring and validation [23]
- Wibed, a platform for commodity wireless testbeds(2013) [24]

4.1 Design

As shown in Figure 4.1, testbeds installations based on the WiBed platform are composed by a set of COTS routers, the *testbed nodes*, forming mesh networks with access to an external *testbed server* reachable through one or more gateway nodes. The testbed management system follows a server-client model with the *testbed controller* (the server software) being the only means of external interaction with the whole testbed (thus, ideally, neither sysadmins nor researchers should ever log in to the nodes). The nodes receive *orders* from the controller (e.g. “install a new experiment”) in a pull-based manner, by periodically sending it a *request*. The orders are embedded in the *replies* the

\(^1\)http://wireless.kernel.org/en/users/Drivers/ath9k  
\(^2\)http://battlemesh.org/BattleMeshV7
controller issues for the node’s requests. Controller orders are node-specific, making it possible, for instance, to stop the experiment execution on a single specific node but also in a set of nodes.

Experiments, as explained in 4.2.1.1, are filesystem overlays which are attached to the nodes firmware during the experiment execution. Each node can run only a single experiment at a time but different nodes can run different experiments in parallel. The management system also allows the execution of commands in the nodes when an experiment is running.

Aside from the experiment deployment tools, WiBed includes a centralised storage system to ease data collection from experiments and an error reporting system for debugging purposes.

In order to relax deployment restrictions, the testbed management and related nodes-controller communication is established over a separate wireless mesh network, operating independently of the wireless experimentation network. Although it is recommended to include more wired nodes in order to increase the testbed resilience, this approach allows to significantly reduce the costly and time-consuming task of deploying a wired network connection in the target experimentation zone. Therefore, if experiments demand low-level WNICs access and the testbed is not fully wired, at least two WNICs are needed to fully isolate the management from the experimentation network.

4.2 Implementation

4.2.1 WiBed Node

4.2.1.1 Filesystem Architecture

Following the OpenWRT approach, the testbed nodes filesystem is composed by two parts: a SquashFS read-only LZMA compressed ROM containing the basic operating system (kernel, a mini-
mal root filesystem and the testbed management software) and a JFFS2 mounted as OverlayFS\(^3\) over the read-only partition to store filesystem changes.

The standard boot process of OpenWRT is as follows\(^4\): The kernel boots from ROM and executes `/etc/preinit` which executes `/sbin/mount_root`. Read-write partition is combined with the read-only partition to create a new virtual root filesystem. Then bootup continues with `/sbin/init` and loads the operating system.

WiBed extends this approach by adding a new component: a second overlay aimed at allocating the experiment placed in an external storage device\(^5\) (such as a USB stick). The experiment overlay is only mounted during the execution of an experiment as figure 4.2 and figure 4.3 show.

To perform an experiment the researcher must provide a set of files (filesystem) which will be copied to the experimentation overlay. The system synchronises the files of the standard overlay with the experimentation one. Finally the system is configured to boot with the new overlay and the node is rebooted.

### 4.2.1.2 Firmware Overview

A single generic firmware image is used for installation in all testbed nodes. To coordinate all the processes running in the node, a local Unified Configuration Interface (UCI) database is used. It is placed on `/etc/config/wibed` and contains some static predefined fields such as WiFi channel, BSSID or node hardware model, and a set of runtime dynamic fields such as status, last command executed or current experiment identifier\(^6\).

In the first boot the node configures itself writing the changes to the internal overlay. It detects the available WNICs and Ethernet interfaces. Then, it configures the management network according

---

\(^3\)Included in Linux Kernel mainline 3.11


\(^5\)The additional storage device overcomes the common space limitation of the internal storage in current COTS routers.

\(^6\)[https://wiki.confine-project.eu/wibed:config](https://wiki.confine-project.eu/wibed:config)
4.2. Implementation

4. WiBed: The Campus Testbed

Table 4.1: Node states

<table>
<thead>
<tr>
<th>stateID</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INIT</td>
<td>Booting</td>
</tr>
<tr>
<td>1</td>
<td>IDLE</td>
<td>Idle (waiting for action)</td>
</tr>
<tr>
<td>2</td>
<td>PREPARING</td>
<td>Downloading overlay</td>
</tr>
<tr>
<td>3</td>
<td>READY</td>
<td>Overlay ready to be installed</td>
</tr>
<tr>
<td>4</td>
<td>DEPLOYING</td>
<td>Installing the overlay and rebooting the node</td>
</tr>
<tr>
<td>5</td>
<td>RUNNING</td>
<td>Experiment running</td>
</tr>
<tr>
<td>6</td>
<td>RESETTING</td>
<td>Resetting the node to its default configuration</td>
</tr>
<tr>
<td>7</td>
<td>UPGRADING</td>
<td>Upgrading firmware</td>
</tr>
<tr>
<td>8</td>
<td>ERROR</td>
<td>Error detected</td>
</tr>
</tbody>
</table>

to the parameters specified in predefined UCI sections. The IP assignment is organised according
the last bytes of the MAC address. Finally, the node starts the pulling process from the controller
and it is ready for experimentation. This way the only manual intervention needed from the testbed
administrators is to configure properly the gateway nodes (a UCI configuration flag must be enabled).

The figure 4.4 shows a diagram that summarises the executed steps of a WiBed node from its initial
boot to the end of an experiment.

Figure 4.4: Simplified functional diagram for a node

4.2.1.3 Management System

The node states are detailed in Table 4.1. Figure 4.5 shows the node finite-state machine with trans-
sections resulting from a controller order tagged. The remaining transitions are the result of a node’s
local operations. RUNNING and RESETTING are the only states where the experiment overlay is
mounted. An unexpected behaviour in the node’s side leads to the ERROR state directly or after a
given number subsequent attempts. Recovery from the error state is formally an internal transition
but can only be triggered externally via the execution of a command.
4. WiBed: The Campus Testbed

4.2 Implementation

4.2.2 WiBed Controller

The WiBed controller works as a standard web server (implemented in Python with the Flask framework) providing a REST API endpoint for node management but also a web interface and a REST API for interaction with the users.

4.2.2.1 Node Rest API

The nodes of the testbed periodically send pull requests to the node API endpoint. The controller parses these requests and stores information about the nodes on a local database, sending only needed information and/or commands in response to a node request. Consequently, the network bandwidth required is optimised. All exchanged messages are made in JavaScript Object Notation (JSON)\(^7\) standard text format (simple, well known and human-readable).

The pull requests correspond to state transition on the node’s side (loop-back transitions included). Requests contain the node status. The controller responds to each request with a reply containing an order such as: create and manage an experiment, perform an upgrade, run a set of commands or the empty order.

The registration of new nodes on the controller is made in a totally reactive manner. When the controller receives an API request containing a new node id, it will consider that request as coming from a new node and will add it to the management database. Nodes are attached to the testbed in the INIT state and they revert back to that state after an upgrade. When nodes are in the INIT state they send their device model and firmware version along with the request. This allows the controller to always know updated hardware details of each node in this ad-hoc management operation. As a result, the delivery of compatible firmware to testbed nodes is ensured and the researchers can choose nodes with similar hardware and up-to-date firmware.

Currently, the servicing of experiment and firmware images is done via a simple HTTP response to a GET request (to static/overlays/\(\text{overlayId}_i\) or static/firmwares/\(\text{firmwareId}_i\)). However, the sys-

---

\(^7\)http://www.json.org/
WiBed: The Campus Testbed

4.2. Implementation

4.2.1 WiBed: The Campus Testbed

The system is designed in a flexible manner to allow future experimentation with other delivery mechanisms such as BitTorrent or wireless-mesh-optimised P2P technologies.

4.2.2 Web Interface

For researchers and administrators a web interface (figure 4.6) acts as a front-end, where the state of the nodes can be checked, experiments can be started/finished, commands can be issued, and experiment or firmware images can be uploaded. Attached to a controller’s response to the node, a set of commands can be added. Commands can be grouped in two categories, the experimentation commands (executed during an experiment by a researcher) and the administration commands (executed by the testbed administrators at any time). The output of the executed commands (both stdout and stderr) are attached to the next communication message to the controller. This way all nodes can be managed from a single point, making individual remote access through SSH unnecessary.

4.2.3 Functionality

- **Admin tab:**

![Figure 4.6: Main page of WiBed web interface.](image)

---

Footnote 8: Secure Shell
This page (figure 4.7) is used to send commands to the routers in the testbed by selecting them manually. More specifically, the server will create as many entries in the database as routers will receive this command. These commands added to be executed will be shown in a list in the bottom of the page showing, not only the information regarding the command and how many routers have executed it already, but also as a link to a command page with the execution information of each router.

![WiBed web interface - Admin Tab.](image)

**Figure 4.7: WiBed web interface - Admin Tab.**

- **Nodes tab:**
  This page (figure 4.8) has a list of all the available nodes in the different testbeds working in the controller showing its identification name, its testbed name, its status and the time since the last pull to the server\(^9\). Each node in the list is also a link to a information page of the router.

- **Node’s page:**
  Each node in the test has an informational page (figure 4.9) with information regarding the node, such as the model, firmware version, last success pull to the server, its status, if it is

---

\(^9\)Administrator users are able to show/hide nodes to the research users.
4.2. Implementation

4. WiBed: The Campus Testbed

Figure 4.8: WiBed web interface - Nodes Tab.

performing an experiment, a list of previous experiments where it was involved and a description. Furthermore, there is an OpenStreetMap section available showing node’s current location and being able to be modified if necessary (by clicking its position in the map or with its coordinates).

- Repo tab:
  This page has a list of firmwares available to be downloaded. First, they are split by the development branch [master (stable version) and last trunk (testing version)]. Once in the development branch, there is another list of router architectures available (WiBed’s current supported architecture is ar71xx, but mpc85xx architecture will be supported soon). Finally, each architecture folder has its own different models’ firmware and also packages already compiled for them.

- Errors tab:
  This page (figure 4.10) has a list of router’s IDs which have gone to the Error state during their operation. The available options are to watch the log files online (a list of log files) or to download them as an id.tar.gz file.

- Topology tab: This page (figure 4.11) contains an automatically generated topology graph of the nodes that belong in the WiBed UPC CN-A deployment. The graph provides information about the existing link between nodes as well as the link quality.

4.2.2.4 Functionality for Researchers

- Experiments tab: This page (figure 4.12) shows the information regarding the experiments being performed in the testbed and the ones finished, as well as it brings to the researcher the possibility to create new experiments.

- Add Experiment page:
  This page (figure 4.13) allows the user to name the experiment, select or add an overlay and the nodes involved in the experiment.

---

10 Administrator users are able to hide/show the node to the researchers and also to delete it from the server.
11 Administrators are able to modify the description of the nodes.
4. WiBed: The Campus Testbed

4.2. Implementation

Figure 4.9: WiBed web interface - Node Page.

- Experiment information page:
  This page (figure 4.14) allows the user to manage the experiment process. The researcher can start or stop the experiment, check general information about the experiment, check the nodes and its status and finally add commands that will be sent to all the involved nodes.

- Results tab:
  This page (figure 4.15) shows a list of finished experiments and its time stamp. Each experiment contains an expanded list with the results of each node involved in the experiment and, as in the error’s tab, the results can be downloaded or checked online.

4.2.2.5 Functionality for Administrators

- Firmwares tab:
  This page (figure 4.16) allows the administrator to add new firmwares to install them in the nodes automatically. Also it shows information about existing firmwares in the server and in which routers these are installed.

- dbDebug tab:
  This page shows the information contained in the database of the server in a sorted manner. This information is informative only and it is not available to be modified.
4.2. Implementation

4.2.2.6 Rest API for Users

A Rest API is under development to allow future integration with common management interfaces in the CONFINE project. Additionally to these services, the WiBed controller provides a repository of WiBed firmwares, an error logging system integrated with the nodes, an upgrading system for the nodes and a storage system where results of the experiments are transferred from the nodes and stored automatically.

4.2.3 Management Network

The WiBed management network is used to connect the research devices between them and with the testbed controller. Wired connectivity between research nodes is not a requirement for all WiBed nodes but for at least one (identified as the gateway). Consequently, to ensure the controller-node communication, the management network has to be built using WiFi 802.11 standards. Thus the primary WNIC radio is used to create an AD-HOC (IBSS) network between all deployed nodes.

The routing protocol BATMAN-ADV\(^{12}\) handles the layer 2 routing by encapsulating Ethernet over Ethernet. As a result, all the nodes are in the same collision domain. This facilitates the management and administration of the nodes, since standard auto-configuration and node access techniques via IPv6 link-local addresses are possible.

The gateways support the interconnection between the WiFi testbed and the controller.

A proper operation of the nodes and experiments is essential for the usability of the testbed. Therefore, to clearly identify and handle the cases of correct and abnormal experiment execution, while coping with potential instabilities of the wireless and multi-hop management network, the system must combine robustness against temporary connectivity-failures with restrictive checks and recovery procedures. Only in case of long-term disconnection and unrecoverable failure, the node automatically returns to the initial state (even if there is an experiment running).

\(^{12}\)http://open-mesh.org
4. Wildes: The Campus Testbed

4.3 Source repositories

The WiBed platform is an open source software package with an associated documentation wiki page and a source repository. Following the CONFINE project methodology, the WiBed source code is hosted in the project’s Redmine server.\(^{13}\)

The current source repositories are:

1. **wibed-controller**: This repository stores the development source of the server-side code. This source consists on the web server and database management.

2. **wibed-openwrt**: This repository stores the base necessary files to compile the WiBed firmware. This base system consists on a frozen snapshot of OpenWRT, stable and further tested, plus the extra repositories needed for the WiBed firmware linked to the compilation process.

3. **wibed-openwrt-routing**: This repository stores a frozen snapshot of the OpenWRT routing repository. It is used as a tested and stable source for the routing protocols used in WiBed.

4. **wibed-owrt-packages**: This repository stores a frozen snapshot of the OpenWRT packages repository. It is used as a stable and tested source for the packages used in WiBed.

5. **wibed-packages**: This is the base repository of the WiBed packages which stores the two branches of packages that make up the firmware.

\(^{13}\text{WiBed Redmine project page: http://redmine.confine-project.eu/projects/wibed}\)

**Figure 4.11**: WiBed web interface - Topology Tab.
4.4 Getting Started with WiBed

The usage of WiBed has been documented in the Confine Wiki, in a page dedicated to WiBed\(^{14}\). This page provides public documents related to WiBed such as research articles, user guides, various usage examples covering different use cases, information how to build your own testbed as well as information concerning the internals of WiBed.

\(^{14}\)http://wiki.confine-project.eu/wibed:start

---

**Figure 4.12:** WiBed web interface - Experiments Tab.

**Figure 4.13:** WiBed web interface - Add Experiment Page.
4.5 Limitations and Future Work

4.5.1 Limitations

The main current limitation for the WiBed platform is the mandatory usage of a specific Linux Kernel (included in the base system). This issue might be solved by using kexec\textsuperscript{15}, a tool which allows the execution of a new Kernel at run time. However using this approach would be dangerous for the testbed stability due to the lack of control and the possibility of breaking the management system (as a result the node would be lost and only a manual intervention could recover it).

Another current limitation is the lack of mechanisms to detect incompatibilities between the man-  

\textsuperscript{15}https://www.kernel.org/pub/linux/utils/kernel/kexec

Figure 4.14: WiBed web interface - Show Experiment Page.

Figure 4.15: WiBed web interface - Results Tab.
4.5. Limitations and Future Work

4.5.2 Current status and further work

Our ongoing work involves supporting more devices (e.g. TL-WDR4900\textsuperscript{16} and ALIX2d\textsuperscript{217}), completing the user Rest API and evaluating the UPC A6 testbed. The integration with Community-Lab at the controller level is a topic that has not yet been addressed because it would require modifications to Community-Lab’s controller.

\textsuperscript{16}http://www.tp-link.com/lk/products/details/?model=TL-WDR4900
\textsuperscript{17}http://www.pcengines.ch/alix2d2.htm
4.5.3 Costs and replicability

As already mentioned, the entire WiBed platform is a free/libre-software project, and thus available to everybody. The total cost of the hardware of the presented 40-nodes UPC testbed is below 3,000 €. The skills required for designing and implementing wireless experiments may suffice to install and operate a WiBed-based testbed. Thus, the solution presented here allows the deployment and execution of a fully operational wireless testbed at a fraction of the cost required by most other available testbeds.

It is worth mentioning here that WiBed can also be used as a quick mesh network deployment solution for specific purposes. For example, the WiBed platform could be used to quickly deploy a mesh network with Internet access where the nodes would be turned into access points. In this case, the user would deploy an experiment that provides the service he wishes to offer around the network.

4.6 Conclusions

This chapter presented WiBed, a platform for deploying and managing testbeds for experimenting on mesh networks built on top of COTS IEEE802.11 routers, extending the existing experimental infrastructure of Community-Lab. We have presented its design, and how nodes evolve throughout the execution of an experiment and react to commands given by a central controller. We have also described how these nodes interconnect to one another and, eventually, to the controller server. By focusing on a very pragmatic and simple ad-hoc operation and management we have achieved to reduce both the budget and effort requirements for setting up of link-layer to application-layer experiments over these wireless testbeds. In addition, the implemented platform is free/libre open source, thus any individual or group of researchers can use it to quick deploy a low cost Also Wi-Fi, related to wireless networking technology, from the slogan: Wireless Fidelity (WiFi) testbed.
5 Common NodeDB

For community wireless networks (CWN), a node database serves as a central repository of network information. This “registry” functionality is separate from the testbed controller, which is described above. The testbed controller manages the CONFINE testbed and the experiments (creation of slivers, slices, etc). In contrast to this, the common NodeDB Common Node DataBase (CNDB) manages the network information per se for the community network.

It is a registry, a link planning tool, an IP address assignment tool, etc. It comprises information about nodes deployed at certain locations, devices installed at these locations, information about internet addresses, and – in networks that use explicit link planning – links among devices.

All this information is maintained via a web or RESTful [25] interface by the community members. Therefore the common NodeDB contains the static as well as the dynamic information about the community network as opposed to the experimental testbed network information. It is easy to see that a node database is a central component of any community network. Usually community networks thrive to be decentralized, however there are a few centralized components which cannot easily be distributed: IP address assignment and information on optimal channel assignments. It helps to have tools such as the NodeDB for planning these common, shared resources in a community network.

An overview of the overall architecture of the NodeDB is given in figure 5.1. It is easy to see that all

![Figure 5.1: Overview of the overall architecture of CNDB](image-url)
static information (“Registries” node top-middle) is used by almost any application or service except for the dynamic information collected by the “Collector” (lower right). All information is available via the “Dashboard”, the central web-based user-interface.

A registry (the Node database called NodeDB in the following), a dashboard and a collector for collecting monitoring and statistics data (one part of it being a “spider” that collects data from web-interfaces of deployed devices) have been implemented during the project. The spider is also used during the conversion process for augmenting the existing information in the legacy database of Funkfeuer Vienna as some of the needed information of the new NodeDB was not available. In addition the monitoring and statistics data was immediately relevant for WP4 (experiments).

5.1 Motivation for a Common Node Database

One goal of the CONFINE project is the integration of research devices into community networks for research purposes. Research devices need some information about the location where they are deployed as well as on the structure and interconnection of the community network. This is needed to allow experimenters to access topology data and to manage large experiments with many research nodes.

To avoid duplicating data in the community network database and the research database, the common node database implementation was started. Starting from the CNML definition and the guifi.net implementation and from node database implementations of Funkfeuer Wien, Funkfeuer Graz, guifi.net, Ninux Italy, WLAN Slovenia, AWMN, an object model was designed and implemented. It has already been informally reviewed by members of some of the CNs mentioned.

Historically almost any new wireless community network has implemented their own database solution. The reason for this is probably that it is hard to understand – and adapt to own needs – other people’s code. When starting a new network, people want to take control of their networking needs and therefore chose to implement their own solution to be in control of the code whenever it needs adaptation to new requirements. There were some efforts to come up with a common implementation of a node database but so far none of these efforts has seen deployment beyond one or two installations.

One of the reasons that up to now no common node database exists is that there are diverging needs of each running network on the one hand, and the need to fully understand (and be able to modify) the software used to run the network on the other hand.

Many different programming languages, web frameworks, and databases are used to implement web applications. Therefore, existing node database implementations use widely different implementation technologies and have – although they share a common set of requirements – focused on different aspects of the daily operation of a community wireless network. This has resulted in a plethora of different solutions to similar problems with different quality of service across the set of solutions offered.

To get a better understanding of the common requirements of community wireless networks and to eventually facilitate interoperability, the community network markup language project was founded in 2006 [26]. To date we know of an implementation by guifi.net and another similar XML export by Athens Wireless Metropolitan Network (AWMN). We also studied several existing implementations and their data models before starting NodeDB.

To test our implementation against requirements of different networks, converters to import data from three different networks were implemented: Two networks are Funkfeuer Vienna and Funkfeuer Graz.
– although both are named “Funkfeuer” they have completely unrelated node database implementations – the third network being the CNML export of guifi.net mentioned above.

5.2 Architecture of the NodeDB

The design of the object model (Figure 5.2) is based on the analysis of several existing node databases of various community wireless networks. It is also the basis of a common API definition. Discussions with developers of CWNs have indicated that the model is compatible enough to existing databases to allow the implementation of the same common API for different CWNs.

The object model can represent requirements of different CWNs, e.g., one CWN may do explicit link planning using a routing protocol where point-to-point links need to be configured explicitly (e.g. the Border Gateway Protocol Border Gateway Protocol (BGP)) while other networks may use a mesh routing protocol like OLSR. An example of the former is guifi.net while e.g. Funkfeuer uses OLSR. The IP-Address reservation supported by the object model can use both, IPv4 and IPv6 addresses and can use different allocation strategies which are different in each network. Funkfeuer Vienna, for example, uses public IPv4 addresses that are allocated to nodes in the network. Funkfeuer Graz and many other community networks use RFC1918 private IPv4 addresses [27] which are not routable on the internet. Naturally IP-address reservation strategies differ for these networks and the differences need to be reflected in the IP-Address pools and the permissions for individual users.

Some notes on the object model: we try to keep only the relevant attributes of a real-world object in the object itself – everything else is modelled as a relation. The blue arrows denote inheritance relationships (IS˙A). The yellow arrows are attributes, e.g., the Node has an attribute manager of type Subject which is required (this is implemented as a foreign key in the database).

The black arrows are 1:N or N:M relationships (also implemented as foreign keys in the database) but the relation objects have their own identity. This is used to separate the attributes of an object from its links to other objects. It also implements referential integrity constraints: a link is deleted if any object it refers to is deleted.

There are different link types. A two-way link (implementing a 1:N or N:M relationship) has a left and a right side (or link role). An example is Wireless˙Interface˙uses˙Wireless˙Channel; in the diagram this link object is displayed as `uses` between the Wireless˙Channel and Wireless˙Interface. The black arrows connecting these are labelled left and right which indicates how this should be read. Note, that in this case the left attribute is on the right side in the diagram. A two-way link like this has an identity and can have additional attributes besides left and right; it also can appear as link role in another link type.

There are also unary links with only a left side. An example is the Device which cannot exist without its left attribute, the Device˙Type. There can be several devices with the same device type. This relationship is inherited by Antenna and Antenna˙Type and Net˙Device and Net˙Device˙Type.

5.3 Application Programming Interface API

As can be seen from Figure 5.1, most applications like the map, link planning or configuration generator access the registry data, the database. The interface used by these applications is a RESTful [25] interface that allows access to the database and to the IP-Address reservation module. It also supports access to metadata like date/time and user of changes to an object.
The API can use tokens for authentication (so-called REST authentication tokens) or client-side certificates.
5.4 Node Database Applications

Traditionally, different CWNs have implemented different applications on top of their own node database. During requirements analysis, several common applications have been identified. Note that we have not implemented all of these applications, but instead we tried to get some existing applications ported to the new common database API. Also note that the list is not exhaustive.

The following items were – at least partly – implemented, some details are provided in the following.

- **Registry**: Register users with their name and email address, register nodes and devices and associate them to users. This is mostly provided by the Dashboard.

- **IP address registry and allocation**: Allow users to register IP addresses or IP ranges for their devices and make sure no address is registered twice. Also allow to de-allocate an existing IP allocation. Since many CWNs run on RFC 1918 private IP addresses, one of the challenges is to move to IPv6 addresses; a new common node database implementation has to support IPv4 and IPv6.

- **Map**: Draw all nodes on a map to get a graphical overview of the network; most existing community networks have implemented a map service.

- **Generation of tools necessary for running a network**, e.g., name-server configuration, directory services.

- **Generation of configuration for network monitoring and -alerting tools** like smoke-ping, nagios, or cacti.

The following applications are planned for the future or may be implemented as experimental features:

- **Link planning**: This answers the question: "If I try to interconnect these two devices via antennae, will it work?". Note, that some CWNs use explicit link planning (where each link needs an explicit configuration in the routing protocol used) while other networks use a dynamic routing protocol to establish links (e.g., OLSR). Both should be supportable by the node database.

- **Auto configuration of devices**: To facilitate deployment of new nodes, some community networks have implemented various degrees of auto-configuration for devices; this makes it easier for users to deploy new devices. In addition to auto-configuration of existing devices, some applications allow the generation of firmware for various devices.

- **Social networking functions**: allow community members to contact other members and invite them to build a link to their node.

- **Federation**: Allow services such as DNS zones to be federated across the network.

- **Services offered by the network or by users**, e.g., Voice over IP or video on demand applications. Note, that in some existing CWNs services like this are not considered an experimental feature but are already deployed in production quality.

5.5 Dashboard

The dashboard supports a user view on the node database that is focused on the tasks users typically want to do when managing their nodes and associated devices.

The dashboard view shows the nodes of a user, the devices and interfaces. From the list of nodes one can be selected and the devices shown are restricted to those associated with the chosen node. The same can be done for interfaces associated with a device. This can be seen in figure 5.3.
5. Common NodeDB 5.5. Dashboard

In addition the dashboard view shows part of the funkfeuer map and life data about the nodes and devices of the user.

For realizing the dashboard as a responsive web application, Bootstrap was used for the first prototype. But since bootstrap changed the whole API between major versions 2 and 3 and the complicated html structure needed, we settled on Pure CSS for the responsive web application framework.

Dashboard forms support auto-completion. This helps to keep the data in the Node database consistent.

*Figure 5.3: Dashboard example*
because users are less inclined to invent creative new (incompatible) spellings for existing objects.
A good demonstration how auto-completion works, and why it is necessary, are the address fields of
the forms for creating or changing nodes.
For some fields, auto-completion is restricted to the value of a single field; for other fields, auto-
completion completes several fields at once. For instance, the zip field of node.address triggers
completion of zip, city and country.

![Figure 5.4: Dashboard example small device](image)

This example demonstrates that completion is a good mechanism to avoid inconsistent spellings of
data. When converting from the old Funkfeuer database one would expect that auto-completion of a
zip code that starts with “113” would result in only up to 10 different zip codes and associated city
and country: Vienna has ten different zip codes starting with 113 from 1130 to 1139. But users have
invented all sorts of different city entries associated with theses zip codes, e.g. they add the district
in Vienna to the city name. So cleaning up the data from the old database to a consistent format is a
major effort in the converter.
In the recent version of the dashboard care has been taken to make the contents displayable on devices
of different sizes. In Figure 5.3 the full display version is shown as it appears on a desktop computer.
The map can be turned off to save space. In addition the layout is automatically changed for a small
device as can be seen in Figure 5.4 which shows the same contents as the larger variant. Not only has
the map been turned off but the layout is changed and some attributes are not shown to save space.
5.6 IP Address Reservation

IP Address reservation was recently extended to support user access permissions, quota and the definition of policy parameters that are used when an address (or address range) is no longer used. When returning addresses, it is often necessary to wait for some time before the address is re-assigned, we call this a cool-down period.

5.6.1 Quota

One requirement is that it is checked if a new address is allocated for the correct purpose – being it a node or network interface. Then quotas should be enforceable, for users (limit the number of IP addresses a single user is permitted to allocate) and node or interfaces (it makes sense to limit the number of IP addresses that can be allocated to a single network interface or to a node).

This needed some changes in the current object model. For existing networks a pool of IP addresses is not limited to a single IP subnet. For example in Funkfeuer Vienna, several ranges of IP addresses are used in the public network. Now if quotas were associated with IP subnets, a clever user could allocate IP addresses from several IP subnets, effectively working around the quota. So we added IP-Pool (for IPv4 and v6) to serve as an umbrella for several IP subnets available for allocation.

The above-mentioned access restrictions (quota and definition which users are permitted to allocate from a certain pool of ip addresses) are now associated with these IP-Pool objects. An IP-Pool can contain several IP Subnets.

Quota definition takes the form of an IP (v4 or v6) netmask. Since not only single IP Addresses can be allocated, we needed a way to specify the number of allocable subnets in a compact way. Network administrators are familiar with the notion of a netmask, so we use this to specify quota. A quota of /30 for an IP4 pool in this notation allows a user to allocate four IP addresses (with a netmask of /32) or two /31 networks or one /30. With this notation it isn’t possible to allow arbitrary numbers of IP addresses (e.g. the allocation of 5 /32 addresses) but the common use-case is to allow a user to allocate a single subnet for IPv6 (e.g. /56) and several IPv4 Addresses – usually one per network interface so this limitation isn’t a problem.

For a pool it can be defined how big the networks that are allocated from it may be (aside from the quota per user). For this purpose we define a netmask range. With this setting, one use-case is to limit the allocation of public IPv4 addresses (as used by Funkfeuer Vienna) to single /32 addresses. Another use-case would be the allocation of IPv6 ranges to a node with a netmask of /56. The limit needs not be a single netmaks but can be a range of netmasks. If a range is used, this can result in fragmentation which is avoided if a single netmask is used.

5.6.2 Freeing of Addresses Previously Allocated

If an IP address (or range) is no longer used, it can be marked as free. The IP-Pool mentioned above does contain a Date interval that defines for how long a returned address is marked as “cool down”. Only after the cool down period has expired is the network returned to the pool for reallocation. A garbage collector has been implemented that searches for addresses with expired cool-down period and returns them to the pool.

When a users reserve sub-network of IP addresses an associated IP-Pool can be defined. A user may define a cool down period but to prevent denial of service attacks (reservation of IP addresses in a subnetwork that expire after a very long time), the effective cool down period is the minimum over
all parents. If no cool down period is found, an address freed is marked with the current date to be returned to the pool on the next run of the garbage collector.

The garbage collector also checks if all addresses marked with an expiration date (after which the cool down period ends) are still consistent with the cool down settings of containing network pools.

5.6.3 Access Control

IP-Pools can have associated access rights. As previously mentioned these can contain quota definitions but the main purpose is to allow certain groups of network users the right to allocate IP addresses or ranges (subnetworks) of IP addresses. For this purpose, persons can be grouped forming a Group object. This Group object is granted access rights for IP allocation.

5.7 Spider

Originally intended for augmenting the data used by the converter from the old node database of Funkfeuer Vienna (see 5.1), a spider was written that extracts the following information from the web-interfaces of the nodes in the Funkfeuer network in Vienna:

- Version and type of software used
- WLAN configuration (if available): channel, signal, ESSID, BSSID, etc
- Network interfaces and configuration information
- IP Address information

The reason we do this is that the new database collects more information about the network than the old database used to. For example the type of network interface – wired or wireless – is not stored in the old database. Apparently it is possible to run a large wireless community network without knowing which links in the network are wireless.

The spider can be used on any network that uses OLSR for routing – currently the spider relies on the OLSR topology data for finding out from which IP addresses to retrieve data.

Funkfeuer currently uses a mix of different hardware and software components. The spider can currently handle the following software on devices:

- Devices with Freifunk firmware
- Devices using the Luci web interface
- Vienna Backfire
- OLSR Textinfo plugin
- Native OpenWRT web interface
- Ubiquity Router OS

The type of firmware running on the device is auto-detected. When testing, the software auto-detected many nodes from the Funkfeuer Graz network (which uses a different set of devices from Funkfeuer Vienna).

We currently spider the network once a day and keep the data retrieved for statistics on network parameters. Some of the data will be made available via our Statistics Server.

With the new spider we found a network-wide configuration error: One IP address in use by a prominent mesh node (with many connections) was re-used by a network interface of some minor node.
5.8 Conclusions and Lessons Learned

Ongoing activity is the community feedback for a rollout in Graz, focusing on Graz first, later on Vienna. The Common Node Database CNDB is now in a state where we have deployed two test instances with real users and real data, one for Funkfeuer Vienna, the other for Funkfeuer Graz. These installations have already got some feedback from the community. The users in Graz will deploy our solution first but feedback has been slow and we expect the deployment after the end of the Confine project.

It turns out that getting feedback from people doing this in their spare-time is more time consuming than initially anticipated. So far we have had several feedback cycles with meetings in Vienna and Graz that resulted in changes to the dashboard and to the configuration of the IP-Address reservation.

When writing importers – for both CNML data from Guifi.net as well as SQL database dumps for Funkfeuer Vienna and Graz – we encountered problems with the data offered. In particular, non-sanitized data (like invalid MAC Addresses and naming variants mentioned in 5.5) and problems with character encodings.

The problems with character encodings were due to the long usage period of the data in question which had encountered changes of character set (from Latin1- to Unicode-based encodings like UTF-8). Some data was double-encoded. This resulted in a module for the SQL dump reader to sanitize the encoding problems.

For sanitation of data in the future (avoid inconsistencies during data entry) we implemented auto completion in the dashboard as detailed in section 5.5.

Maintaining the Spider in a fast-changing network turned out to be no easy task – adding parsers for new web interfaces like the recent addition of Ubiquity Router OS is time-consuming. On the other hand the information from the spider – like the detection of a duplicate IP address in use (see 5.7) – is invaluable. We therefore plan to keep the spider running to cross-check static configuration data in the node database against the deployed configuration in the network.
6 Testing Framework

6.1 Introduction

Automated testing of the software being developed was introduced in November 2013 to improve the release chain of CONFINE software. It allows the developers to check that software behaves as expected, and test certain bug fixes without introducing new ones. In this chapter, we will describe the architecture and the impact of the testing framework on the development, complemented with a short description of the various components provided to the developers and a "how to" for running the tests manually. During the project the test suite was incrementally expanded by adding new tests for new features and new tests to increase the test code coverage, leading to a full testing suite to guarantee stability and quality.

What follows is a final overview of the testing framework, based on experiences running the testing system throughout the entire project. Final numbers are given where possible.

6.2 Architecture

In this section we describe the architecture used for the automated testing as shown in Figure 6.1. The

![Figure 6.1: CONFINE automated testing architecture](image-url)
6. Testing Framework

6.3 Tests

Tests are made of atomic action blocks followed by atomic verification blocks. The following guidelines were used to create the tests.

An atomic action block:

- Should represent a single action the research actor can take on the system under test.
- Should be a wrapper for the represented action. There should be hardly any additional logic in this procedure. Any additional logic could introduce bugs unrelated to the system to be tested.
- Should follow the same API as used by the research actor. Parameter checking should be done by the system under test, not by the atomic actions.
- Will probably be a mapping of the REST API and command calls via ssh to the nodes.
- Returns whether it’s wrapped call succeeded or not (via boolean value or exceptions).
- Should not verify whether the call actually did what it was supposed to do: this is done by the verify blocks.

A compound action:

- Should be a serial list of atomic actions which the research actor can take on the system under test. If additional logic in this list is needed, having separated compound actions might be preferred.
- Describes a single use case by taking atomic actions without additional logic.

An atomic verification block:

- Allows a test to verify an action on the system under test.
- Should represent a single boolean condition that the researcher can take on the system under test.
- Is most likely a wrapper around a GET call on the system under test.
- Should verify the get value of the system under test to the expected value (or within range of expected values)
6.4. Jenkins interface

In this section we will briefly describe the different components in the Jenkins interface. We refer to the help files and document of Jenkins for functionality that is not covered here.

6.4.1 Overview

Each Jenkins job and its state can be found in the Jenkins overview page as depicted in Figure 6.2. The state depends on the result of the tests. A green sphere means all tests passed, yellow indicates some tests failed, and a red sphere indicates that something went wrong and the system was unable to be tested. The column “Test Result” shows the results of the last run executed for that job. Within the brackets, the difference with the previous test run is shown. The last column shows the the time since the last successful run of the job. We will describe each job shortly.

**FirmwareBuilder Newstyle** This job is responsible for generating and keeping the testbed node software up to date. It is triggered by the VCTBuilder job such that the correct node software version is bundled in the VCT container.

**Tests** Only API and VCT tests are executed on an emulated Community-Lab testbed.

**Tests Coverage** In this job, all tests are executed in an emulated Community-Lab testbed while keeping track of the code that is executed. It provides the developers with the test coverage. This way, untested code can be discovered and tested.

**Tests Django** There are dedicated Django tests for the controller. These are executed and the results stored in this job.

---

- Should not perform the action to fulfil the condition: this is done by the action blocks.
- A compound verification:
  - Should be a serial list of atomic verifications similar to the compound actions.
  - Should be a set of verifications that all should be fulfilled to pass the compound verification. Thus the AND of all the atomic verifications is the result of the compound verification.
- **A test:**
  - Should have a list of dependent tests. These are tests that should be successful before running this test. 
  - Should first have a set up. It uses atomic or compound actions to create the scenario for testing. The actions used in the build up should not be verified because the test depends on the tests of the actions used in the build up. Verification of those actions is done in the dependent tests.
  - Should then execute a testing action. This is the action for which the test was designed.
  - Should at last verify the testing action.
  - Should not have complicated additional logic.
  - Should return whether the test succeeded or not (boolean value/exceptions)
  - Should restore the system when either the test fails or succeeds.

Often the CRUD approach was taken to test various components. The abbreviation CRUD stands for create, read, update, and delete which are the actions taken on the objects in various scenarios.

The tests are written in Python and can be found in the `confine-tests` repository, available in the CONFINE repositories.
6. Testing Framework

6.4. Jenkins interface

Tests Installation Different installation scenarios within the VCT container are tested in this job.

Tests Integration The integration tests are small scaled tests using the testbed. In these tests, nodes are allocated, slices and slivers created, and a validation test is run within the slivers. They are a great resource to get familiar with the controller and node API.

Tests ORM The CONFINE-ORM [35] software is separated tested in this job.

Tests Real Controller Besides testing on the emulated Community-Lab testbed, a real testbed is maintained with real hardware nodes. This job executes the tests on this real testbed.

Tests Strict Some tests trigger an issue within the software that is considered non-important. This job executes these tests with the API and VCT tests on an emulated testbed.

Tests VM Controller This is another real testbed, however, the nodes contained in this testbed do not run on dedicated hardware but are virtual machines.

VCTBuilder All tests using the VCT container trigger at some point this job. It is responsible for generating and building the correct VCT container.

6.4.2 Test matrix

Most jobs consist of a configuration matrix. This is a matrix containing all possible combinations of different software versions. Using conditional statements, some options can be eliminated if they make no sense. An example matrix is shown in Figure 6.3. Here, two versions of the CONFINE software (master and testing) are tested in combination with three versions of the node firmware software (master, testing, testing˙axel) and in combination with two versions of the controller software (origin/master, origin/testing˙snt). This results in two times three times two, thus twelve different VCT containers to run the tests to. In this example, the controller branch origin/testing˙snt is temporally disabled to reduce the number of combinations to test.

Deliverable D2.7
6.4. Jenkins interface

6.4.3 Test results

For each job, the test result trend is shown, similar as in Figure 6.4.

The colour green represents the passed tests, yellow the skipped, and red the failed tests. This graph provides a quick test result overview of all the archived previous runs, showing the progress made. For each test configuration matrix entry, the latest test results are shown as in Figure 6.5.

It shows, for an entry in the matrix, the number of tests that failed (red), skipped (yellow) and passed
(blue) and the difference in numbers with the previous test run. For the failed test, the assert that failed can be found in the collapsed entries of the table named All Failed Tests. Each failure has a duration and an age. Often these table entries are used as reference to the outcome of a tests in bug reports.

### 6.4.4 Test coverage

One of the jobs is the test coverage where the coverage of the tests on the code under test is measured. It generates a graph similar to Figure 6.6.

![Figure 6.6: Jenkins’ test coverage](image)

The graph gives an overview of the test coverage for previous test runs. The test coverage is divided in packages, files, classes, lines and conditional coverage. It provides a great tool for the developers to focus the creation of tests.

### 6.5 File system structure

We will, in this section, briefly present each file found in the `confine-testing` repository.

- `./dl` This directory is used to store downloaded files by the script.
- `./researcher` In this directory, the LXC container containing the researcher’s filesystem is extracted. It also contains the patches for the Researcher container.
- `./sshkey` The ssh keys, both private as public, are available here to log in the Researcher LXC container.
- `./vct` The VCT LXC container is extracted by the scripts in this directory. It also contains the patches for the VCT container.
- `./vct-container` This directory is used to create a new VCT LXC container.
- `host.sh` This script is used to configure the host running the two LXC containers. It contains functions to set up network bridge and masquerading for the LXC containers and is used by `test.sh`.
- `researcher-container.sh` This script can be used to generate a new Researcher container including additional software.
6.6 Statistical Evaluation of Impact

In August 2014, the project evaluated the impact of testing by means of a statistical analysis, explained in what follows. The results of this analysis show that the testing has a positive impact, which confirmed to the consortium the decision to continue testing throughout the rest of the project.\textsuperscript{1}

6.6.1 Bug reports

In August 2014, the CONFINE test suite contains a total of 143 tests that cover 75\% of the lines of code. This is a very decent coverage considering the size of the project. The testing framework was introduced in November 2013 and has been aiding the developers for about ten months in August 2014. As shown in Table 6.1, the number of bug reports mentioning the testing framework for the CONFINE controller accounts already for 28\% of the bug reports in only ten months. For the CONFINE distribution, the bug reports created from the testing framework account for 6\%.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Reports mention test framework</th>
<th>Total #reports</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>confine controller</td>
<td>54</td>
<td>193</td>
<td>28%</td>
</tr>
<tr>
<td>confine distribution</td>
<td>10</td>
<td>166</td>
<td>6%</td>
</tr>
</tbody>
</table>

Table 6.1: The relationship between the tests and bug reports.

However, we divided the bug reports in the set of pre test framework reports, these are the reports filed before November 2013, and the post test framework reports, as shown in Table 6.2.

Here, the rate of bug reports per month (brpm) changed for the CONFINE controller from 3.7 reports a month to 15.2. This is five times as many bug reports a month. Considering only the post test framework reports, the testing framework is mentioned in 36\% of the reports since its introduction. The first three months, the contribution of bug reports mentioning the testing framework accounted for 36\% of the reports.

\textsuperscript{1}For reasons of efficiency this analysis was not repeated in year 4. However, the reported positive effects have been noted by developers again in the final project year, without a firm analysis.
Table 6.2: The relationship between the tests and bug reports pre and post the introduction of the test framework.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Pre test framework</th>
<th>#Post test framework</th>
<th>Reports rate (bug reports per month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>confine controller</td>
<td>41</td>
<td>152</td>
<td>3.7 to 15.2</td>
</tr>
<tr>
<td>confine distribution</td>
<td>102</td>
<td>64</td>
<td>6.8 to 6.4</td>
</tr>
</tbody>
</table>

for 25%. The last three months, this increased to 45% as the testing framework matured and the developers relied more on this system.

For the CONFINE distribution project, the bug report rate did not changed since the introduction of the testing framework. However, out of the 64 bug reports since the introduction, 10 reports mention the testing framework, accounting for 16% of the bug reports.

### 6.6.2 Resolved bugs

Looking at the number of bugs resolved since the introduction, 47 out of the 54 mentioning the testing framework for the CONFINE controller are resolved, this corresponds to 87%. Before the introduction of the testing framework the rate of resolved bugs for the CONFINE controller was around 0.82 resolved bugs a month and thus lower than the rate of new filed bug reports (3.7 brpm). Since the introduction, this increased to 16 resolved bugs a month, exceeding the 15 brpm, for which the testing framework is mentioned in 29% of those resolved bugs.

Also for the CONFINE distribution project we find an increase in resolved bug rate from 2.9 resolved bugs a month (less than the 6.8 brpm) to 7.9 which is greater than the 6.4 brpm. For the latter, 10% mentioned the testing framework. These results can be found in Table 6.3.

Table 6.3: The relationship between the resolved bug rates and the open bug rate.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Pre resolved rate</th>
<th>#Post resolved rate</th>
<th>Open bug rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>confine controller</td>
<td>0.82</td>
<td>16</td>
<td>2.88 to -0.8</td>
</tr>
<tr>
<td>confine distribution</td>
<td>2.9</td>
<td>7.9</td>
<td>3.9 to -1.5</td>
</tr>
</tbody>
</table>

As demonstrated, the resolved bug report rate exceeds the corresponding new bug report rate meaning that since the introduction of the testing framework, more bugs are resolved than filed in a month. In other words, the rate of open bugs is a negative rate since the introduction of the testing framework. We can conclude that the testing framework had and still has a major impact in software quality, especially on the CONFINE controller and we showed the importance and need for a testing framework.

### 6.7 Test Running and Requirements

Sometimes a developer wants to locally isolate a test and run this test manually. In this section we describe the requirements for a system to run the tests and the steps to execute them on that system.

#### 6.7.1 Requirements

To run the tests, the system needs the following minimal requirements.

**Internet access** This is needed for downloading the VCT and Researcher container.
Quilt [36] software  Used for the patch management of the containers.

SSH software  This allows the system to initiate the test in the Researcher.

IPv6 support  The CONFINE software is designed to use IPv6.

Conntack support  Both containers are given Internet access through masquerading for which conntack support is needed.

LXC enabled kernel  This support is needed to run the LXC containers.

Mounted cgoup  A requirement needed for LXC.

Git software  Downloading the latest releases of the tests and code, the git software is used.

A 64 bit system  The researcher container is a 64 bit LXC.

6.7.2 How to

Running a test is done as follows. The first step is to checkout the confine-testing repository:

```
git clone "http://git.confine-project.eu/confine/confine-testing.git" confine-testing
```

This repository includes all the necessary scripts to create the VCT and researcher LXC containers. Then cd into this directory:

```
cd confine-testing
```

The test.sh script uses a few variables which can be set or the default values can be used. The variables are:

- **VCT_CONTAINER**  this defines the version of the VCT container to be used for testing: currently this is set to `vct-container,vct˙testing,controller˙origin˙master, nodefw˙testing.tar.xz` by default. The list of available containers can be found at [37].

- **VCT_LXC**  this is the name of the VCT container. Default `vct˙$(date -u +%s)`

- **RESEARCHER_LXC**  this is the name of the researcher container. Default `researcher˙$(date -u +%s)`

- **SETUP_ONLY**  when set to `y`, the script will only set up the VCT and researcher containers and then exit. This way, `lxc-console` for the researcher can be used to manually start a test. Default `n`.

- **NO_SETUP**  when set to `y`, the script will not set up the VCT and researcher containers. This allows you to reuse currently running containers. Default `n`.

- **NO_TEARDOWN**  when set to `y`, the script will not teardown the VCT and researcher containers. This allows for further investigation after the tests. Default `n`.

- **INSTALLATION_TEST**  when set to `y`, the script will install a new VCT inside the researcher container and use this for tests. Default `n`.

- **RESEARCHER_OVERLAY**  when set to `y`, the script will copy the contents for the map `../researcher˙overlay` to the researcher. This allows customising e.g. network settings. Default `n`. 
The only thing which you will want to change is the VCT CONTAINER. A new VCT container is continually generated, triggered by new VCT, controller and node firmware releases. To limit disk usage, only the 20 last releases are kept. Better check quickly in the Jenkins installation for the latest generated VCT CONTAINER:

- Go to http://testing.confine-project.eu/job/Tests/lastCompletedBuild/console
- Click on the branches you want at the bottom of the output (e.g. ConfineDistBranch=master,origin/master,NodeFirmwareBranch=testing)
- Click on the last yellow build in the left column (i.e. the most recent build which has not failed)
- Click Console Output. There you should see a line describing the used container (e.g. Using vct: vct-container,vct3e32990,controller1726c74,nodefw6357d5a.tar.xz).

This can be used as follows:

```
sudo bash -c ""
export VCT CONTAINER=vct-container,vct testing,"
controller origin master,nodefw testing.tar.xz; "
export VCT LXC=vct;  "
export RESEARCHER LXC=researcher;  "
bash test.sh"
```

The script will download, unpack, patch and start the VCT container provided by the VCT CONTAINER variable. It will then download the researcher container and unpack, patch and start it. The CONFINE tests and utilities are automatically downloaded. It will then ssh into the researcher and execute the tests. All this takes a while. After the tests are done, the VCT and researcher containers are stopped.

The IP address of the VCT container is default fdf6:1e51:5f7b:b50c::2/64, for the researcher container fdf6:1e51:5f7b:b50c::3/64 and for the bridge (vmbridge) on the main host fdf6:1e51:5f7b:b50c::1/64.

### 6.7.3 Run tests manually

Make sure that the testing framework is set up using:

```
sudo bash -c ""
export VCT CONTAINER=vct-container,vct testing,"
controller origin master,nodefw testing.tar.xz; "
export VCT LXC=vct;  "
export RESEARCHER LXC=researcher;  "
export SETUP ONLY=y;  "
export NO SETUP=n;  "
bash test.sh"
```

The SETUP ONLY and NO SETUP variables define respectively whether you want the test to be just set up (e.g. to inspect the LXC containers) or you do not want any setup of the containers (e.g. to test local applications or to reuse an existing configuration).

Entering the VCT and researcher LXC containers is done by using their name defined by VCT LXC and RESEARCHER LXC. The login:"vct", the password:"confine" without the quotes.
lxc-console -n $VCT\_LXC
lxc-console -n $RESEARCHER\_LXC

When running the tests in the researcher container, the following command can be used to run all the tests:

```bash
env CONFINE\_SERVER\_API='http://vctserver/api' CONFINE\_USER='vct' 
CONFINE\_PASSWORD='vct' PYTHONPATH=confine-utils:confine-tests 
python -m unittest discover -s ./confine-tests/
```

For a specific test, the following command can be used:

```bash
env CONFINE\_SERVER\_API='http://vctserver/api' CONFINE\_USER='vct' 
CONFINE\_PASSWORD='vct' PYTHONPATH=confine-utils:confine-tests 
python ./confine-tests/confine/tests/test\_group.py
```

For a specific test case, the following command can be used:

```bash
env CONFINE\_SERVER\_API='http://vctserver/api' CONFINE\_USER='vct' 
CONFINE\_PASSWORD='vct' PYTHONPATH=confine-utils:confine-tests 
python -m unittest confine.tests.test\_node.GroupTests.test\_get\_groups
```

### 6.7.4 Run an installation test

The testing framework can also help you test the installation of a new controller. It will start a new installation from scratch, inside the researcher LXC, and then run the tests against this controller. To trigger this, just run

```bash
sudo bash -c " 
export INSTALLATION\_TEST=y; 
export TEST\_API=1; 
export TEST\_VCT=0; 
export TEST\_INTEGRATION=0; 
export UPGRADE=$upgrade; 
bash test.sh;"
```

### 6.7.5 Control which tests to run

As the tests are run automatically in different environments, some variables are defined to control which tests to run. They can be defined similar to the other variables, e.g. the CONFINE\_USER variable. Currently the variables are:

- **TEST\_VCT** run inside VCT environment, so start creating nodes etc. You should turn this on, default value is 0 (no VCT).
- **TEST\_API** run basic REST API tests. You can leave this parameter, default value is 1 (do run API tests).
6. Testing Framework

6.8. Overview of testing resources

To summarize the testing efforts, table 6.4 gives an overview of testing resources online:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Location</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing repository</td>
<td>git.confine-project.eu/confine/confine-testing.git</td>
<td>All code to execute tests</td>
</tr>
<tr>
<td>Tests repository</td>
<td>git.confine-project.eu/confine/confine-tests.git</td>
<td>The tests themselves</td>
</tr>
<tr>
<td>Jenkins</td>
<td>testing.confine-project.eu/</td>
<td>Continuous testing reports</td>
</tr>
<tr>
<td>Documentation</td>
<td>wiki.confine-project.eu/testing:start</td>
<td>Extensive documentation</td>
</tr>
<tr>
<td>Test containers</td>
<td>media.confine-project.eu/vct-container/test/</td>
<td>LXC testing containers</td>
</tr>
<tr>
<td>Test guidelines</td>
<td>wiki.confine-project.eu/testing:guidelines</td>
<td>Test writing documentation</td>
</tr>
</tbody>
</table>

6.9 Conclusion

In this section we presented the testing framework set up to test the CONFINE software components. One of the strengths of the framework is the ability to test the software both with emulated testbeds as well as real testbeds. Then, we demonstrated briefly the interfaces made available to the developers to analyse and follow up the test reports of different version combinations. The impact of the framework was analysed by the statistics of the bug reports for the different software components. This analysis showed unambiguously the importance of the testing framework. Finally, the steps to execute the tests on an emulated testbed are presented as a reference.

With this testing framework in place, the project managed to achieve higher stability, leading to a more sustainable code base.

Deliverable D2.7
7 VCT/ns-3 Integration

In Virtual Confine Testbed (VCT) a researcher can test their experiments prior to deployment in the real Confine Testbed. VCT is designed to be as close to the real testbed as possible. It is an LinuX Container (LXC) Container inside of which each node is implemented as a virtual machine on a server system. The software and firmware used in VCT is almost identical with the one in the real testbed. However, generating a realistic topology with an adequate wireless error model presents a challenge. The out of the box connectivity solution of VCT consists of a bridge that connects all nodes and provides a perfect link. Since this solution is often insufficient for thorough testing of applications designed for wireless environments, a simple link error model based on IPTables and EBTables has been developed. Using the EBTables subsystem a topology can be created by preventing node pairs from seeing each other. Moreover, if combined with IPTables, it allows for configurable lossy links between pairs of nodes. The setup used in Confine employed real measurements of IEEE 802.11 packet loss in a function that estimates the unicast and broadcast packet loss based on the distance between the communication partners. A detailed description of this approach can be found in [38]. Nevertheless, this link error model cannot emulate the effects of collisions or delay caused by transmission speed or retransmissions, and might thus be insufficient for many applications. Therefore some work has been done to integrate the widely used network simulator ns-3 into Virtual Confine Testbed. ns-3 is the successor of the well known NS2, an open source software widely used by the networking research community. ns-3 in addition supports unique features required for the transparent integration of real and emulated resources and it is able to emulate a wide range of components of the communications protocol stack. ns-3 delivers realistic wireless channel models, ways to define network topologies and a real time scheduler. It also integrates well with LXC Containers through its TapBridge module. Thus, ns-3 is an ideal candidate for the provision of real-time link layer emulation to the Virtual Confine Testbed. When the underlying simulation server is fast enough and the real-time constraints are met within the simulation, even quantitative results can be obtained from the experiment.

In this deliverable, mainly the technical aspects of our approach and integration issues with VCT are addressed. For additional information please refer to deliverable [39]. In addition, [39] also provides best practice examples of using the ns-3 enhanced VCT tools for a practical exercise in a Mobile Communication lecture. This does not only offer a unique hands on experimentation environment to students, it also creates interest in and prepares them to perform experiments within the real testbed.

7.1 ns-3 installation

A script for automatic installation of a tested version of NS3 along with the necessary dependencies is provided with the VCT Container. The script can be found in listing 7.1. First, it downloads the necessary dependencies. Since the integration is done inside the container, the script is adjusted to the environment that is known and well defined and doesn’t need to have a generic form for different distributions. After installation of the dependencies, ns-3 sources are downloaded, built and installed. Then the ns-3 builtin tests are executed. Finally custom patches are applied. These consist of a custom large scale fading model and a modification to the TapBridge module. Both are described in more
7. VCT/ns-3 Integration

7.2. Connectivity design

In the original ns-3 code for TapBridge module, a raw socket is assigned the same MAC address as the TAP device it is connected to. This results in wrong addressing when slivers are communicating over ns-3 emulated wireless ad hoc channel. To fix this problem the original code has been modified in such a way that a raw socket is assigned a MAC address of its sliver counterpart. This MAC learning method is implemented by interpretation of ARP requests and responses transmitted through the socket.

7.2 Connectivity design

Since the nodes in VCT are Kernel-based Virtual Machine (KVM) virtual machines instead of LXC containers, the connectivity method had to be adjusted in order to integrate ns-3 into the VCT Container. An example of a connectivity setup with two nodes in one slice is depicted in figure 7.1. Each sliver is assigned an (isolated) interface called wlan0 directly connected with an interface in the node and its KVM counterpart (vnet) on the outside of the node. Like in the real testbed, multiple slivers on the same node are separated by VLAN tags with VLAN IDs that identify the slice a sliver belongs to.

Unfortunately, ns-3 doesn’t support VLAN tagging. Therefore, a VLAN device adjusted to the VLAN ID of the relevant slice is created for each node. It is responsible for removing VLAN tags from outgoing and attaching them to incoming packets. In the case of the setup in figure 7.1, two slivers of

---

**Figure 7.1:** Example of connectivity between two slivers over ns-3 inside the VCT Container.
a slice with VLAN ID 256 are taking part in the emulation. For each sliver, a TAP device is created that is connected directly with ns-3 using the TapBridge module. The VLAN devices are connected with their TAP counterparts through dedicated bridges. Raw sockets on the ns-3 side are assigned MAC addresses of the wlan0 interfaces of the slivers that use them.

In the default setup of the VCT Container, the vnet interfaces of the KVM nodes are connected to one common bridge. The mapping of a vnet interface to a particular node as well as to the common bridge needs to be looked up, before the interfaces can be removed from the bridge and the ns-3 connectivity as described above can be established. To simplify this process for the user, a script has been provided, that checks the interface-to-node mapping by examining XML configuration of the KVM machines. It then destroys the default connectivity setup and creates all the devices necessary to create connectivity between the slivers over an ns-3 emulated channel. Listing 7.2 contains the most vital parts the script - retrieving the mapping with the KVM management tool virsh, creating the ns-3 connectivity as well as restoring the default one. The user can specify the name of the node’s direct interface to be used as parent for the sliver’s isolated interface (eth1 or eth2) and VLAN ID of the slice. Default values appropriate for most use cases have been provided.

### 7.3 Usability adjustments

The web frontend of the Confine controller doesn’t allow for detailed network configuration of the isolated interfaces on the slivers. Thus a user needs to configure every sliver manually by e.g. logging onto each sliver and executing network configuration commands. In order to do this, the IPv6 management address of each sliver must be looked up. For automation of the task we prepared a tool that allows logging onto or executing commands on a sliver, without the direct knowledge of the management addresses. Listing 7.3 contains the code of an auxiliary tool, that utilises the Confine REST API to query for the management addresses of the slivers for a given slice. The output are the IPv6 addresses of all slivers of the given slice in the numerical order of the IDs of the nodes the slivers are deployed on. This information is then used by the tool in listing 7.4. The user can identify the sliver by the slice ID and the number of the sliver as output by the auxiliary tool. Execution of a command on all slivers of a slice can be done using this tool in a simple bash loop.

Since the modified MAC learning mechanism depends on ARP messages being exchanged, it is recommended to clear the ARP cache of every sliver prior to starting an emulation. The tool mentioned above can be used for automation of this task as well.

#### Listing 7.1: NS3 installation script

```bash
#!/bin/bash

PATCHPATH="/home/vct/confine–dist/ns3/patch/"

echo "Installing necessary tools..."
sudo apt-get update > apt.log
sudo apt-get install uml–utilities vtun mercurial gcc g++ python patch >> apt.log

echo $?

echo "Downloading NS3..."

cd
mkdir ns3
```
7. VCT/ns-3 Integration

7.3. Usability adjustments

```bash
ret=$?
if [ $ret -ne 0 ] ; then
echo "Error: directory ns3 exists. If you wish to reinstall ns3, please remove the directory first."
exit $ret
fi
cd ns3
hg clone http://code.nsnam.org/ns-3-allinone -r 2c860e2e968f

ret=$?
if [ $ret -ne 0 ] ; then
  echo "Error: Failed to download NS3. Aborting..."
  exit $ret
fi
echo $ret
echo "Installing and patching NS3..."

cd ns-3-allinone
./download.py
./build.py --enable-examples --enable-tests
cd ns-3-dev
./test.py
patch -p1 < $PATCHPATH"ns3-rice-propagation-v3.diff"
patch -p0 < $PATCHPATH"tap-bridge-cc.diff"
patch -p0 < $PATCHPATH"tap-bridge-h.diff"

cd
```

Listing 7.2: Setup NS3 connectivity

```python
#!/usr/bin/python

import subprocess
import xml.etree.ElementTree as ET
import argparse
def get_nodes_info():
    """ Get a list of ids of all running nodes """
    print "Querying KVM for running nodes..."
    virsh_cmd = ["virsh", "-r", "-c", "qemu:///system"]
    list_cmd = ["list"]
    # get running domains (nodes)
domains_str = subprocess.check_output(virsh_cmd+list_cmd)
    # parse running domains
domains = parse_domain_str(domains_str)
    # sort numerically
    nodes = sorted(domains, cmp=hexstr_compare)
    print "Currently running nodes:" , ", ".join(nodes)
    """ Now for each node get info about it and store in a dictionary """
    print "Querying KVM for detailed node information..."
    for nodeID in nodes:
        print "NODE", nodeID
dominfo_cmd = ["dumpxml", "vcrd"+nodeID]
        xmldump = subprocess.check_output(virsh_cmd+dominfo_cmd)
        root = ET.fromstring(xmldump)

Deliverable D2.7
```

63
node.cnctvty[nodeID]={'eth1':None,'eth2':None}
for iface in root.iter('interface'):
    mac = iface.find('mac').get('address')
br = iface.find('source').get('bridge')
ifname = iface.find('target').get('dev')
print 'IF:', ifname, 'MAC:', mac, 'BR:', br
if br == 'vct−direct−01': eth='eth1'
e elif br == 'vct−direct−02': eth='eth2'
else: continue
node.cnctvty[nodeID][eth]=(ifname, br)
print "...done."
def create_connectivity(vlan, iface):
    print "= Creating NS3 connectivity using following configuration:"
    print "= VLAN ID:"; vlan
    print "= Node direct interface:"; iface
    i=0
    for node in sorted(node.cnctvty.keys()):
        ifname, br = node.cnctvty[node][iface]
        print "************** Processing node", node, "**************"
        try:
            print "Deleting %s from bridge %s..."%(ifname, br)
            subprocess.check_call(delif+[br, ifname])
            print "OK"
            print "Creating necessary bridges and interfaces..."
            subprocess.check_output(ipadd+[ifname, ifname+"."+str(vlan)]+ipvlan+[str(vlan)])
            print subprocess.check_output(tun+["−t", 'tap'+str(i)])
            subprocess.check_output(addbr+["br−"+str(i)])
            subprocess.check_output(addif+["br−"+str(i), ifname+"."+str(vlan)])
            subprocess.check_output(addif+["br−"+str(i), "tap"+str(i)])
            subprocess.check_output(ifconf+["br−"+str(i), 'up'])
            subprocess.check_output(ifconf+[ifname+"."+str(vlan), 'up'])
            print "OK"
        except subprocess.CalledProcessError, e:
            print "====="
            print "Failed while calling:"; e.cmd
            print "Return Code:"; e.returncode
            print "Command output:"
            print e.output
            print "======"
            print "It is recommended to use the script with --restore to revert the changes."
            print "Aborting..."
            return
        i+=1

def restore_connectivity(vlan, iface):
    print "= Reverting changes done for NS3 connectivity using following configuration:"
    print "= VLAN ID:"; vlan
    print "= Node direct interface:"; iface
    i=0
    for node in sorted(node.cnctvty.keys()):
        ifname, br = node.cnctvty[node][iface]
        commands = [ ifconf+["br−"+str(i), 'down'],
                    ifconf+["tap"+str(i), 'down'],
                    ifconf+[ifname+"."+str(vlan), 'down'],
                    delbr+["br−"+str(i)],
                    ...]
ipdel+[ifname, ifname+"."+str(vlan)]+ipvlan+[str(vlan)],
tun+["−d", "tap"+str(i)],
addif+[br, ifname]]

print "*************** Processing node", node, "***************"
print "Removing added bridges and interfaces..."
for cmd in commands:
    try:
        print " ".join(cmd)
        subprocess.check_output(cmd)
        print "OK"
    except subprocess.CalledProcessError, e:
        print "======"
        print "Failed while calling:", e.cmd
        print "Return Code:", e.returncode
        print "Command output:"
        print e.output
        print "======"
i+=1

Listing 7.3: Extract management addresses of slivers

#!/usr/bin/python
import json
import urllib
import sys

slice_data = urllib.urlopen(slice_url, proxies={})

slice_json = json.loads(slice_data.read())

for slivers_url in slice_json['slivers']:
    sliver_local_data = urllib.urlopen(slivers_url['uri'], proxies={})
    sliver_local_json = json.loads(sliver_local_data.read())

    node_local_url = sliver_local_json['node']['uri']
    node_local_data = urllib.urlopen(node_local_url, proxies={})
    node_local_json = json.loads(node_local_data.read())

    node_mgmt = node_local_json['mgmt_net']['addr']

    sliver_remote_url = "http://"] + node_mgmt + "]/confine/api/slivers/" + sliver_local_json["uri"].split('/')[-1]
    sliver_remote_data = urllib.urlopen(sliver_remote_url, proxies={})
    sliver_remote_json = json.loads(sliver_remote_data.read())

    for interf in sliver_remote_json['interfaces']:
        if interf['type'] == "management":
            print interf['ipv6_addr']

Listing 7.4: Log onto or execute command on sliver

#!/bin/sh

PATH="/home/vct/confine-dist/ns3/utils/get_sliver_ips.sh"

Deliverable D2.7
if [ "$#" -lt "2" ]
then
  echo "ssh sliver.sh <slice-id> <sliver-index>"
  echo "ssh sliver.sh <slice-id> <sliver-index> <command>"
  exit 1
fi

i=0
for ip in '"{PATH} $1' do
  if [ "$i" == "$2" ]
  then
    ssh -i /var/lib/vct/keys/id_rsa root@$ip $3
    exit 0
  fi
  i=$((i+1))
done

i=$((i-1))
echo "Illegal sliver index, must be between 0 and $i"
exit 1

7.4 Conclusions

Our approach of combining VCT with ns-3 emulation offers an additional way of testing experiments in a more realistic network setting before deploying it in the real testbed. It offers more detailed emulation for link characteristics and a broader variety of communication models than our simple link error model based on IPTables and EBTables. Thus, it provides an additional option between using the vanilla VCT and the real testbed. VCT with ns-3 can even provide quantitative experiment results when certain conditions are met to guarantee real-time characteristics.

Overall, the ns-3 enhanced VCT enhances the experimentation workflow, especially for testing experiments that need realistic link properties and a non-trivial network topology. It opens doors to test other non IEEE802.11-based communication technologies and thus supports an evolution of Community Networks. In addition, it offers a more controlled environment then the real testbed and thus can be a valuable addition for the benchmarking environment (see [39] for details). Furthermore, it offers the potential of extending the real testbed with virtual resources.

Concrete next steps can be a tighter integration of our scripts to connect VCT with ns-3 into the VCT Web interface. In addition, for some cross layer experiments it is crucial to get link layer information from the emulated communication resources. Here, a DLEP implementation within ns-3 can allow for an exchange of this information between the emulated radio devices and the slivers running the experiments.
8 Dynamic Link Exchange Protocol (DLEP)

One focus of the CONFINE project is to improve software tools of Community Mesh Networks. Fraunhofer FKIE has improved and extended the OLSRv2 implementation of Olsr.org to provide Community Mesh networks with a better access to the second generation OLSR protocol – OLSRv2. The OLSRv2 implementation uses a protocol called DLEP of the IETF MANET working group to get physical and linklayer data from an external radio attached via Ethernet.

DLEP is a protocol that standardizes a local connection between a bridging radio and a router to allow the router to learn about the known linklayer and physical layer data (e.g. bitrate, frequency or signal strength). All DLEP control data is sent on the same interface the user data traffic is sent. DLEP contains an UDP based auto-discovery mechanism that establishes a TCP session between radio and router to exchange a set of data both sides agree upon during session initialization.

The Fraunhofer FKIE implementation contains two extensions for DLEP with additional data types. One defines additional data types for physical layer data like signal strength, frequency or bitrate. The other one defines additional link-layer data types, mostly statistics like number of send and received frames, lost frames and retransmissions.

8.1 Implementation

The IETF MANET working group DLEP draft changed substantially during year 4 of CONFINE. Between draft version 7 [40], which was the base of the first Fraunhofer FKIE implementation, and draft version 16 [41] all identifiers and length fields of the protocol were extended to 2-byte values. In addition to this the negotiation of additional data types between DLEP-radio and DLEP-router were changed from a “per data type” to an extension system. These two changes conflicted with some basic assumptions of the existing DLEP code and triggered a complete rewrite of the code to allow more sharing of code between DLEP-radio and DLEP router and increase the modularity of the DLEP extension implementation.

8.2 Integration

Fraunhofer FKIE uses OpenWRT based DLEP-radios as direct interfaces for its local testbed. The DLEP radio is configured (see listing 8.1) without any knowledge of the VLAN based sliver separation.

The main issue of DLEP-radios in combination with the Confinie Sliver architecture is that the DLEP radio does deliver its link layer data within the datapath of the DLEP-radio, but not within the VLAN-tagged traffic that is available to the Slivers.

To resolve this problem, Fraunhofer FKIE implemented a DLEP-proxy (see 8.1) for the host system of the research node which receives data with a DLEP-router instance on the Ethernet connection to the external DLEP-radio and provides this information to all Slivers through a DLEP-radio attached to the internal bridge of the Research Device (see listing 8.2).
8.3 Conclusion

Listing 8.1: DLEP radio configuration on radio

```plaintext
config global
  option 'failfast' 'no'
  option 'pidfile' '/var/run/dlep_radio.pid'
  option 'lockfile' '/var/lock/dlep_radio'

config log
  option 'syslog' 'true'
  option 'stderr' 'true'

config dlep_radio
  list 'name' 'wlan0'
  option 'datapath_if' 'br−lan'
  option 'not_proxied' 'false'
  option 'proxied' 'true'

config nl80211_listener
  option 'if' 'wlan0'
  option 'interval' '1.0'
```

Figure 8.1: DLEP proxy in Confine architecture

A custom extension of the DLEP-radio and DLEP-router implementation of Olsr.org allows the user to split the DLEP control-plane from the data-plane and map them to two different interfaces. This provides an easy way to integrate the DLEP-provided information into the Slivers (see listing 8.3).

8.3 Conclusion

The integration of DLEP into the CONFINE testbed allows for additional link-layer based tests without the need of a direct access for Slivers to the Wifi driver and all the security and privacy issues that are connected to this access. The DLEP integration was also used as a testbed to see if the current implementation is powerful and flexible enough for users. This pushed the implementation from a ”dlep-like but not compatible” early implementation through a DLEP-07 compliant one to the current DLEP-15 compatible implementation, which in turn created useful input for the development of the DLEP standard at the IETF.
The next steps for DLEP are to implement the newest draft release -16 to prepare for the discussion (and following changes) of the security issues raised by a few working group participants. While there is a rough consensus that heavy security would be counterproductive to some DLEP features among the DLEP design team, there is still more work necessary to form a consensus together with the security area directors, which could block DLEP from becoming a Standard Track RFC.
9 Testbed monitoring system

The Community-Lab monitoring system was designed specifically to monitor activity on the Research Devices (RD) of the Community-Lab testbed. In this section, we present the motivation behind this monitoring system and details about its design and metrics.

9.1 Motivation

Monitoring the Community-Lab testbed presents specific challenges that the designed system should address:

- Large scale of infrequently used data.
- The monitoring system should support active measurements that provide insight into the functioning of a node without revealing too much information of what is running on it.
- It should also gather slice and sliver specific information.
- The system should be flexible enough to add new metrics without hampering already existing functionality.
- Monitoring logs should never lose precision and should support passively measured data such as last-time SSH succeeded, number of ports in use, resource hogs (which experiments are using the most CPU, memory, bandwidth and ports).

A general-purpose monitoring system does not meet these special-purpose requirements of Community-Lab and are meant for different workloads and properties. Slice specific and sliver specific information (lxc monitoring) cannot be obtained directly by any of the existing monitoring systems. Nagios, Zenoss, ntop, Ganglia and cacti use RRDtool (Round Robin Database tool) for storing data. RRDtool is great for storing time series data and aggregating information, but it is quite inflexible. It becomes necessary to compromise between flexibility and efficiency. Adding new metrics would require updating the database file (RRA). Once an RRA (Round Robin Archive) is created, it is possible to change existing values and add new data sources, it is not possible to add or remove metrics and change their properties. If modelling of data is not considered carefully, it can lead to a number of updates as and when new slivers are created in a node. Slice specific data implies data from different nodes and would result in a dynamic list of RRD which in turn would need additional scripts to fetch, aggregate and display data. For instance in Comon (monitoring system of PlanetLab), the data model is carefully chosen, but still old database files are deleted when the format changes. In many cases (depending on configuration) if an update is made to an RRD series but is not followed up by another update soon, the original update will be lost. This makes it less suitable for recording data such as operational metrics. There is no way to back-fill data in an RRD series and depending on the data model, a single RRD receiving data from multiple sources can be affected by this. Given the large scale varying resource consumption and the dynamic nature of Community-Lab, flexibility is a key requirement.

Apart from that, sliver-centric information is not easily integrated into node-centric data provided by off-the-shelf monitoring systems. This kind of data gathering is also part of the motivation for
developing a separate monitoring system to meet the specific needs of Community-Lab.

### 9.2 Design

At a high level, the monitoring system consists of:

- A **monitoring client** running on each Research Device.
- A **monitoring server**, located in the Community-Lab Controller that provides centralised data gathering and a processing infrastructure.
- A **display facility** to visualise the monitored metrics.

Figure 9.1 shows an overview of the communication model used by the Community-Lab Monitor. The monitoring system designed follows a *Receiver-Pull Model* where the monitoring server pulls the monitored information from the research devices. The client daemon on each RD actively monitors certain system specific metrics periodically. The information monitored by the clients is pulled periodically (every 5 minutes) by the server through a parallel request sent to all Community-Lab RDs. The core design of the monitoring system was not changed during year 4.

![Figure 9.1: Receiver-Pull Model followed by the Community-Lab Monitor](image)

### 9.3 Monitoring Client

The client part of the monitoring system has two main elements:

- A *daemon* that monitors the required information.
- A *micro web server* to allow the monitoring client to accept HTTP queries.
Following we will provide a more detailed information about such elements.

### 9.3.1 Client Components

As we can observe in Figure 9.2, the daemon running on the Research Device monitors the OS and other sources of the node to provide node-centric data (Host Monitoring), such as CPU usage, memory and network data. In addition, it monitors each one of the Linux Containers (LXC) present in the node to provide Sliver specific information (LXC Monitoring) and monitors periodically (every 60 seconds).

The micro web server component allows the monitoring client to accept HTTP requests and responds with HTTP responses, to allow the monitored information to be accessed from web browsers in addition to being used with automated systems. To allow researchers to query and use this monitored data, the response to an information request from the monitoring server is provided in JSON format. This is done by means of a REST API, as shown in the figure.

The daemon stores the monitored information in a log file locally until the data gathering service has seen it. This ensures that no monitored information is lost during a network partition and helps researchers diagnose any problem that may have happened during this period.

It is important to notice that the monitoring client supports the addition of pluggable daemons that can be easily integrated and allow to monitor additional data that is not collected by default by the monitoring client.

---

![Figure 9.2: Components of the Monitoring Client located in the Research Device](image)

---

### 9.3.2 Client Specifics

As mentioned previously, the monitoring daemons in the RD perform monitoring tasks every 60 seconds and store the monitored information in log files. This data is collected from the daemons using a pull model and is fetched every 5 minutes by the monitoring server.
Following we will explain some details about how the monitoring client manages the monitored information to be able to deliver it to the server when it makes a request. More specifically, we will explain how the log files and timestamps work.

9.3.2.1 Log Files

The monitoring daemons on the RDs perform monitoring tasks every 60 seconds. In addition, the clients have to respond to the server every time they receive an information request (generated every 5 minutes). Consequently, the monitoring clients have to perform the following basic operations:

1. Collect node-centric and sliver-specific data.
2. Generate a unique sequence number to unequivocally identify this data.
3. Parse the monitored information, including the sequence number and the collected data to a JSON format and store it in a log file.
4. Keep track of the most recent sequence number of the monitored information.
5. After receiving a request of monitored information by the server, the monitoring client must:
   a) Return the data in a JSON file. This file will only contain the collected data whose sequence number is between the most recent sequence number and the last seen sequence number of the last request received from the server.
   b) Delete all the entries of the local log files whose sequence number is smaller than the last seen sequence number of the last request received from the monitoring server.

Figure 9.3 depicts how the monitoring daemon manages these sequence numbers.

9.3.2.2 Timestamps

The log files were the monitoring data is stored also have several timestamp fields:

- A local timestamp \( T_m \) or monitored timestamp containing the local time on the Research Device measured when a particular monitored information was collected.
9.4 Monitoring Server

The server side of the monitoring system has three main components:

- A **front end** that manages the users’ requests and delivers the metrics collected by the Community-Lab monitoring system.
- A **middleware** that performs all the data collection, storage and processing.
- A **back end** containing a database with all the monitored information.

Following we will provide a more detailed information about such elements.

9.4.1 Server Components

Figure 9.5 shows the different components that are part of the monitoring server: The front end, the middleware and the back end.

The **front end** manages the users’ requests and provides all the technologies required for the visualisation of the monitored data. Monitored information is reported via a web interface that supports sorting, and shows graphs of historical data. It receives HTTP requests, manages HTML webpages and sends the information to the users’ web browsers.

Significant changes were introduced to the front end in year 4. The front end was redesigned to provide a responsive and user interactive environment along with support for sorting and searching RDs according to various metrics. A screen-shot of the home screen is shown in figure 9.6. All entries marked in blue allows further navigation with detailed information.

The **middleware** receives information requests from the front end and fetches the corresponding data from the database. It has three basic elements:
9. Testbed monitoring system

9.4. Monitoring Server

1. A collect module responsible of collecting every 5 minutes the information locally monitored by all the Community-Lab Research Devices.
2. A store module that is in charge of storing the data collected by the collect module in the database.
3. A process module whose main task is to fetch data from the database, stored as JSON files, and deliver it in a format readable for web interface, more specifically, HTML files. Due to the

Deliverable D2.7
fact that the database does not support time series, this module is responsible for reading raw information stored in the database and convert it into time series data. This allows the front end to display historical data at the web interface.

Finally, the back end is a CouchBase Server containing all the monitored information. It is a document-oriented, NoSQL database that stores the monitored data in JSON documents.

### 9.4.2 Server Specifics

In this section we explain some details about the information stored in the CouchBase database located at the back end of the monitoring server.

#### 9.4.2.1 Document types

As mentioned before, the CouchBase server is a database that supports the storage of JSON files. In our case these files will contain the information monitored by the Community-Lab monitoring system. Accordingly, we will have four basic types of JSON documents:

1. **Node**: This kind of files contain the node-centric data monitored by the client daemons running on the Research Devices.
2. **Synthesized**: This kind of documents contain data monitored by the server (more specifically, by the collect module).
3. **Traceroute**: This type of files contain the information about the network topology collected by a pluggable client daemon running on the Research Devices.
4. **Most recent**: These documents contain the most recent data stored in the database. They are copies of the most recent documents. For every one of the three types of documents explained above, the server will generate a copy of the most recent ones. Therefore, if we take into account the most recent documents, we will have six different types of documents stored in the database.

Figure 9.7 shows an example of how the monitoring server generates the most recent documents from the node documents collected from all the Research Devices. As we can see in the figure, for all the JSON documents of type node generated by an RD, the server creates another document of type node_recent containing a copy of the last node document received.

#### 9.4.2.2 How documents are stored

As explained in the Monitoring Client section, the RDs monitor node-centric data every 60 seconds. This data is saved in a JSON format, which means that the RDs will generate a JSON document every minute. This information is stored in the database of the server using an unique identifier composed by the IPv6 of the RD node and a timestamp corresponding to the moment when the monitoring server receives the information sent by the RD. Figure 9.8 depicts an example of a node document generated at time $\Delta$ by an RD with IP “IPv6”. This document is received by the server at time $\Delta'$ and stored in the database using the document identifier IPv6-$\Delta'$.

Equally, for any type of JSON document generated by the client or by the server, it will have a document identifier composed by the IPv6 of the node that generated the document and a timestamp corresponding to the moment when the monitoring server receives it.
9.5 Monitor Metrics

Monitored information is reported via a web interface that supports sorting, and shows graphs of historical data. The reporting currently covers OS-provided metrics and metrics synthesized from other sources on the node.

Deliverable D2.7
When a user accesses the homepage of the Community-Lab monitoring system, they can visualize the monitored information as shown in Figure 9.6. This information is fetched by the homepage from the most recent documents that are copies of the corresponding “node” and “synthesized” document types.

Logs from node-centric data and other metrics monitored by the server are stored in the database. This information is aggregated and summaries are provided at a granularity necessary to make meaningful inference from the data. Precision of the monitored information is never lost and it supports data offloading which is then provided as an open data set.

9.5.1 Node Metrics

The daemon running on the research device provides node-centric data including sliver specific information and monitors periodically (e.g. every sixty seconds). Slice-centric information is generated by analyzing the node-centric logs and is stored in the database. The client daemon accepts HTTP requests and responds with HTTP responses, to allow them to be accessed from web browsers in addition to being used with automated systems. The response is provided in JSON format to allow researchers to query and use monitored data.

Figure 9.9 shows the node type document in JSON format, received by the server at time $= 1385076755.63$ and containing node-centric and sliver specific data of the RD with IPv6 = fdf5:5351:1dfd:9b::2. As we can observe, the name of the document is therefore, a combination of the IPv6 and the timestamp: [fdf5:5351:1dfd:9b::2]-1385076755.63.

The system reports the following OS-provided metrics:

- **Uptime:** It is a measure of how long the system has been on since it was last restarted. Is the total number of seconds the system has been on until last boot.

- **Network data:**
  - Total number of bytes sent and received.
  - The number of bytes sent and received during the last second.
  - For each one of the network interfaces on the RD we also have the total number of bytes sent and received and the number of bytes received the last minute.

- **Average load:** This metric is relative to the number of cores in the RD’s CPU. For example, if we have 2 cores and load_avg = 2, this implies that the system is running at maximum capacity and therefore the CPU is occupied all the time. If load_avg > 2 it means that the system is overloaded and there are tasks waiting for the CPU.
  - $Load_{avg\_15min}$, $load_{avg\_5min}$, $load_{avg\_1min}$ how many tasks are waiting or running in the CPU in the last 15 min, 5 min and 1 min respectively. These metrics help us to get the history of the CPU load in the last minutes and have and idea of its evolution.
  - Tasks scheduled to run.
  - Total number of tasks.

- **Virtual memory utilization:**
  - “available” is the total memory space available in bytes.
  - “total” is the total memory size in bytes.
  - “used” is the total memory space used in bytes.
  - “percent_used” is the total percentage of memory space used.
9. Testbed monitoring system

9.5. Monitor Metrics

- **Disk:**
  - “size” is the disk size in bytes.
  - For each disk partition we have the total size in bytes, the space used (in bytes and in percentage) and the free disk space in bytes.

- **CPU utilization:**
  - “num cpus”: is the number of cores in the CPU.
  - For each processor we have the percentage of CPU usage.
  - The total percentage of CPU utilization.

- **Slivers:**
  - For each sliver running on the RD we have its state and metrics of the memory and CPU utilization.
  - We also have the slice name to which the specific sliver belongs to.

- **Timestamps:** We have three different timestamps in the node documents.
  - “server_timestamp” is the local time of the server when it receives the JSON document.

---

**Figure 9.9:** Monitored metrics in the Research Device
with the monitored information.

- “monitored timestamp” is the local time on the RD when the client performs monitoring tasks.
- “relative timestamp” is calculated by the client daemon (see the Timestamp section of the client daemon). It allows the server to know the relative time interval between all the different node documents generated by a specific RD.

- In year 4, an additional node metric that monitors the state of the RDs was added and presents the error state of the RD if any through the front end.

The relative timestamp allows us to visualize in the monitor homepage not only the most recent information (from the most recent JSON documents, but also the node metrics evolution over time. As an example, figure 9.10 shows the CPU usage over time for a Research Device.

![Node: [fd5:5351::1fd::6f::2], Value: Total CPU usage](image)

**Figure 9.10: Average CPU usage**

Also, the status of slivers along with their IP address and resource usage is shown in Figure 9.11. The system maintains only a manageable set of metrics that help the researchers get insight of any strange behaviour in a given node. To facilitate the researchers in selecting nodes to run their experiments, the web interface provides a Treemap view of all the nodes based on the historical trend (customizable) of resource usage. Figure 9.12 shows a Treemap view of all the monitored research devices in the testbed.

### 9.5.2 Server Metrics

The synthesized metrics are a special kind of metrics monitored by the Monitoring Server instead of by the Monitoring Client daemons located in the RDs. Synthesized data for each Research Device includes:

- Last time the monitoring daemon on the RD was seen.
- Open ports.
- Traceroute to the RD.
- Ping status.
- Slice centric information.
After collection, this information is stored in the Server’s database as a “synthesized” type JSON document. Figure 9.13 shows an example of the content of a Synthesized document containing information monitored by the server at time 1383842050.4 about the RD with IPv6 address fdf5:5351:1dfd:99::2.
9.5. Monitor Metrics

9.5.3 Network Topology

The Community-Lab monitoring system also visualizes the topology of the nodes that are part of the Confine network. This network topology metric is useful to understand the relationships between the different Community-Lab nodes, that is, how the Research Devices are related to each other. Figure 9.14 shows the information displayed by the monitoring system after collecting the network topology metrics. We can observe the Controller node in yellow, the different RDs in green and other network nodes in light blue.

This new metric is extremely important for researchers that want to deploy experiments in Community-Lab. By visualising the Network Topology, an experimenter can easily detect, for example, if all the Research Devices that he was going to select to run his experiments are connected to the same Community Device (CD). This visualization of the network map facilitates the experimenter to take advantage of the distributed nature of the community networks while deploying their experiments.
experiments and avoid the pitfall of selecting multiple RDs that are connected to the same community device. Figure 9.15 depicts an example of the Network Topology of nodes connected to the same CD.

Figure 9.15: Example of selection of RDs connected to the same CD

In order to determine the topology of the Community-Lab network we add a Traceroute Pluggable Daemon to the monitoring clients located in the different RDs. Figure 9.16 shows how the RDs containing these daemons send traceroute packages every 60 seconds to the network node that hosts the Community-Lab Controller. The Monitoring Server then receives all the traceroute messages from all the RDs belonging to the Community-Lab network and responds accordingly. This way, all the Traceroute Daemons collect then information about the path from the specific RD to the Controller.

Figure 9.16: Traceroute pluggable daemon

The information collected by the Traceroute Daemons is saved in a JSON document and sent to the Monitoring Server to be stored at the database. These JSON files are the “Traceroute” type documents mentioned in the Server Specifics section. The content of these documents is depicted in Figure 9.17. As we can observe, the Traceroute document saved by every RD contains a “trace” or list of the nodes that are in the path from such RD to the Controller. For each node we have the following information:
9.5. Monitor Metrics

Testbed monitoring system

Figure 9.17: Traceroute metrics monitored by the server

- IP address.
- Name.
- Round-Trip Time (RTT).

For clarification, let's consider a network composed by 6 nodes, where 1 of them is the Controller and only 2 of the others are RDs. Figure 9.18 shows an example of the list of nodes that each RD would return to the Monitoring Server.

As we can observe in the figure, each RD saves a list containing the hop sequence from such RD to the Controller. When the Monitoring Server located in the Controller performs monitoring tasks every 5 minutes, it collects this information in form of JSON documents and stores them in the CouchBase.
database. The Middleware component of the server then fetches the hop sequences for each RD from the database and converts them to direct neighbour relationships. According to the example in the figure, we miss information about node 5, but this fact is not a problem for us because we are only interested in having information about RDs and their relationships but we do not need to have a precise representation of the network topology.

9.5.4 Inter-RD Connectivity

In year 4, support for network information between RDs was implemented. This includes information such as packet loss, latency, number of hops and degree of connectivity between every RD in the network. The implementation also provides an easy visualisation of this data in two forms: Single RD connectivity and network matrix.

The home screen of the front-end system shows a tab called “connectivity” for each RD as shown in figure 9.6, which upon navigation leads to a visualisation shown in figure 9.19.

![Confine Research Device Status](image)

**Figure 9.19:** Example connectivity of a particular RD to other RDs in the network. Link length is characterised by the latency between the RDs

The RD we are interested in monitoring the connectivity is shown in yellow and the RDs that are connected to it are shown in green. The other RDs that are in the network but have no connectivity are shown in red. The link length of the connectivity is representative of the latency between the RDs. Longer link length means a higher latency. The visualisation is interactive and upon selecting any node, information about the link quality is presented on the screen. The implementation also significantly eases the need to search for any RD in the graphical interface with the help of a “select a node to focus” drop down. All the RDs are listed in their alphabetical order and selecting any one automatically focuses the visualisation towards the RD of interest.

Single RD connectivity is designed to get detailed information of connectivity in the form of link length for each RD separately. The global network quality information is visualised through a network matrix as shown in figure 9.20.

The implementation also allows for easy ordering of the global information based on alphabetical ordering of RD names, RD ordering based on latency, packet loss and node degree. Hovering over any box provides detailed information about that link. The colour intensity is a visual cue to the link.
9.6 Queries and Open Data

In this section, we explain the process that any Community-Lab user can follow to make personalized queries to the monitoring server and to use the information retrieved to generate open data sets.

9.6.1 How to Make Queries

To be able to make queries you will need access to the CouchBase server. You can ask for a username and password by sending an email to the confines-devel mailing list. Then, you will have full access to the CouchBase server through the following URL: http://monitor.confine-project.eu:8091/index.html.

Following we will explain how to make queries to the CouchBase server database and retrieve information about specific monitored metrics.

Views is the method used by CouchBase to process the information stored in the CouchBase Server database to allow data indexing and querying. Views basically creates indexes on the information allowing to search and select information stored within the CouchBase Server. In our case, Views are used to process the different JSON document types mentioned previously to obtain specific monitored metrics from a particular node or set of nodes and time range. This way, we facilitate and automate the data querying process for later representation or analysis. To process the information, Views perform MapReduce operations over the JSON files to return key-value pairs.

Figure 9.21 shows an example of the basic operation of views in CouchBase. As we can see, the output of a View call are key-value pairs, being the key the name of the JSON file and the value the file itself. In this specific example, the output keys corresponds to the node identifier and the values are null and therefore, the resulting JSON files do not have any content but are only pointers to the
The general format of a query using views is the following: **View name(arguments)**

In order to query a view, the view definition must include a map operation that uses the emit() function to generate each row of information. The content of the key that is generated by the emit() provides the information on which you can select the data from your view.

When querying a view, the key can be used as the selection mechanism. We can use any of the following options:

1. **Explicit key**: (key) — show all the records matching the exact structure of the supplied key.
2. **List of keys**: (key_a, key_b, key_c) — show all the records matching the exact structure of each of the supplied keys (effectively showing key_a or key_b or key_c).
3. **Range of keys**: (key_a, key_b) — show all the records starting with key_a and stopping on the last instance of key_b.

Some of the most relevant query arguments are the following:

- **key** (used for “explicit key” selection mechanism): Is an optional string used to return only documents that match the specified key. Key must be specified as a JSON value.
- **keys** (used for “list of keys” selection mechanism): Is an optional array. If set to true, it will return only documents that match each of the keys specified within the given array. Key must be specified as a JSON value. Sorting is not applied when using this option.
- **startkey** (corresponds to “key_a” used for “range of keys” selection mechanism): Is an optional string. It returns records with a value equal to or greater than the specified key. Key must be specified as a JSON value.
- **endkey** (corresponds to “key_b” used for “range of keys” selection mechanism): Is an optional string. If included in the query, it will stop returning records when the specified key is reached. Key must be specified as a JSON value.
9.6. Queries and Open Data

- **reduce**: Is an optional boolean. If it is set to “true” the query will use the reduction function.
- **group**: Is an optional boolean. It is used to group the results using the reduce function.
- **group level**: Is an optional numeric that specify the group level to be used
- **limit**: Is an optional numeric used to limit the number of the returned documents to the specified number
- **descending**: Is an optional boolean used to return the documents in descending key order

The output from a view will be a JSON file containing information about the number of rows in the view, and the specific view information.

When a view is called without specifying any parameters, the view will produce results according to the following criteria:

- Results match the full view specification (i.e., all documents are output according to the view definition).
- Are limited to 10 items within the Admin Console, unlimited through the REST API.
- A reduce function is used (if defined in the view).
- Output items are sorted in ascending order.

It is important to notice that if we use a reduce function in a view definition, we will not have a key value in the resulting output unless we also use grouping.

The format of a general key used in a query to obtain all the information related with an specific node would be:

```
["node id","year","month","day","hour","minute","second"]
```

An example of a view call using this type of key would be:

```
view_name(["[fdf5:5351:1dfd:1::2]","2014","06","24","10","18","25"])
```

The result from this call would be the JSON file containing all the metrics of the node, whose node id = [fdf5:5351:1dfd:1::2], monitored on the 24th of June 2014 at 18 minutes and 25 seconds past 10.

If we use grouping, the group level will be determined by the order of each one of the fields of the key. According to the previous format, group level 1 corresponds to the node id, group level 2 corresponds to the year, group level 3 corresponds to the month and so on.

It is also important to notice that we can omit some fields of the key using “” and therefore, the output will include all possible values from the lowest to the highest. We can also use “{}” to specify the highest value. To get any range of values, we will always need 2 keys, the startkey and the endkey. So when if we want the whole range of values, then the startkey is empty (“” ) and the endkey is (“{}”)

An example of a view call to get a range of values would be the following:

```
view_name(startkey=["[fdf5:5351:1dfd:1::2]","2013","06"," "],
endkey=["[fdf5:5351:1dfd:1::2]","2014", "{}"])
```

In this case, the result from this call would be a set of JSON files containing all the metrics of node [fdf5:5351:1dfd:1::2] monitored from June 2013 (from lowest to highest) and also all the metrics of the same node monitored in 2014 (up to the highest metric monitored).

To explain the specific querying method used by the Community-Lab monitor, we will differentiate between single and set metrics. We refer to single metrics as those whose components are static and
do not change from node to node. Examples of single metrics are: CPU utilization or Load. On the other hand, set metrics are those that could have different number of components for different nodes. Examples of these kind of metrics are: network or disk because we can have different number and kind of network interfaces for different nodes, and similarly, we can have different number and types of disk partitions. For more information about the monitor metrics, please visit the Node Metrics section.

9.6.1.1 Single Metrics

The general format of a key used for querying a single metric is the following:

```
[“node id”, “single metric”, “year”, “month”, “day”, “hour”, “minute”, “second”]
```

An example of a view call using this type of key would be:

```
view_name([“[fdf5:5351:1dfd:1::2]”, “cpu_usage”, “”])
```

The output of this view call will be all the cpu usage values of node [fdf5:5351:1dfd:1::2] starting from the first year monitored up to the last one.

Figure 9.22 shows an example of how views work for this kind of metrics. The example shows a view called get_node-id. As we can observe, the emit function is returning as key the field nodeid of each one of the most_recent JSON documents. In this case, the values are null and we are not using a reduce function.

![Figure 9.22: Example of the get_node-id view](image)

Figure 9.23 shows another example of a view call. In this case we are using the view get_node-cpu_usage_percentage. As we can observe in the figure, the view returns as key the node id and the
timestamp. In addition, the values returned is the CPU usage percentages of the node monitored at each specific time corresponding with the particular timestamp.

Figure 9.24 shows an example of a view using a reduce function to calculate CPU usage statistics from all the nodes. It is important to notice that in this case the output of the view does not have any key. We only have a JSON file containing the returned values after applying the view, which corresponds with all the historical statistics of CPU usage percentage.

Figure 9.25 shows an example of the previous view where we are using a reduce function but we are also using a level 1 grouping. In this case the output of the view has keys even though we are using a reduce function. According to the format of the keys for single metrics presented above, using a level 1 grouping means that the output of the view is grouped according to the node id (corresponding to the first field of the keys for single metrics).
Figure 9.25: Example of the view `get_all_nodes-cpu_usage_statistics` using group level 1

Figure 9.26 shows a similar example but using level 5 grouping. In this case, the output of the view will be grouped according to the hour field of the keys.

Figure 9.26: Example of the view `get_all_nodes-cpu_usage_statistics` using group level 5

### 9.6.1.2 Set Metrics

The general format of a key used for querying a set metric is the following:

```
[“node id”, “set metric”, “single metric”, “year”, “month”, “day”, “hour”, “minute”, “second”]
```

Where, “set metric” can be of two different types:
9.7 Overview of monitoring resources

1. **Network interface**: the field **set metric** can be the name of any of the node’s network interfaces (e.g., wlan0, tunl0, bond0, etc.)

2. **Disk partitions**: here **set metric** can be the name of any of the node’s disk partitions (e.g., /dev/loop1, /dev/sda3, rootfs, etc)

An example of a view call using this type of keys would be:

```
view_name(["[fdf5:5351:1dfd:1::2]","bond0","bytes_recv_last_sec",""])
```

In this case the output of the view will be the number of bytes received the last second by the network interface bond0 of node [fdf5:5351:1dfd:1::2].

Figure 9.27 shows an example of the output of a query of a set metric using views.

![Figure 9.27: Example of the view output for set metrics](image)

### 9.6.2 How to Get Open Data Sets

As explained in the previous section, the result of making a query in CouchBase using views are key-value pairs, where the keys correspond to identifiers of the resulting JSON files and the values are the content of such files.

The fact that the output of a CouchBase query is presented in JSON format makes very easy the process of downloading data from the server, publishing it in an open data format and deleting it from the server database afterwards. The JSON format is a readable format, where all the data is presented as key-value pairs, and therefore, is very easy to convert to any format desired.

### 9.7 Overview of monitoring resources

To summarize the testing efforts, table 9.1 gives an overview of monitoring resources online:
Table 9.1: Overview of monitoring resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Location</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor web interface</td>
<td>monitor.confine-project.eu/</td>
<td>Common entry point for all visualizations, table of current resource consumption per node</td>
</tr>
<tr>
<td>Resource usage</td>
<td>monitor.../visualize/</td>
<td>Visualizations of aggregate and historical resource usage for CPU, memory, network</td>
</tr>
<tr>
<td>Slice list</td>
<td>monitor.../allslice/</td>
<td>List of all slices with links to resource consumption</td>
</tr>
<tr>
<td>Network topology</td>
<td>monitor.../networktrace/</td>
<td>Network graph from the controller over the management network</td>
</tr>
<tr>
<td>Inter-connectivity graph</td>
<td>monitor.../allinternodetrace/</td>
<td>Network connectivity from a node of choice over the management network</td>
</tr>
<tr>
<td>Inter-connectivity matrix</td>
<td>monitor.../allinternodematrix/</td>
<td>Matrix of connectivity among RD by name, degree, latency, or packet loss</td>
</tr>
<tr>
<td>Query interface</td>
<td>monitor...:8091/index.html</td>
<td>Couchbase interface for queries</td>
</tr>
<tr>
<td>Documentation</td>
<td>wiki.../monitor:intro</td>
<td>Documentation about the monitoring service</td>
</tr>
</tbody>
</table>

9.8 Conclusion

In this section we presented the monitoring framework set up to gather and provide detailed information about what is going on in the testbed about nodes, slices, slivers, the underlying network, with precise information, historical data, a query interface and ways to generate open data sets.

With this monitoring framework in place all the Community-Lab testbed is more transparent and accountable about what resources are available, used, environmental conditions, its chronological evolution. Therefore all stakeholders — such as resource providers, consumers or operators — in Community-Lab can be aware of the status and activity in the testbed along time. Despite the naturally uncontrollable nature of a testbed embedded in production networks, this data-rich environment leads to a more informed choices and a predictable and controllable experimentation environment.
10 Conclusions

This document presents the results on the software architecture, design and development of the Confine testbed software system achieved during the fourth year of the project along with an overall summary of the final systems. Some efforts that started previously have continued, while other efforts are developments that allowed to improve the services provided by the testbed.

The main results are:

- **Overall architecture:** Presents an integrated view of the architecture of the testbed facilities, from a network perspective and from a component (and functional) perspective.

- **Core testbed components:** The main focus of recent work was on integrating usability enhancements and stability fixes for an improved usage of the testbed.

- **WiBed: The Campus Testbed:** Recent work has focused on refinements of the WiBed node and the WiBed controller.

- **Common NodeDB:** Development of additional features of the Common NodeDB, in particular the user interface through the dashboard, support for IP address reservation and enhanced functionality of the Spider tool.

- **Testing framework:** The testing framework is based on the Jenkins platform. Recent work has focused on refinements of the testing framework to incorporate more tests and improvements in the documentation.

- **VCT/ns-3 Integration:** This work provided an enhancement to the VCT tool to enable more realistic network and link layer conditions, which improves the testing features that VCT virtual testbed offers to researchers.

- **Dynamic Link Exchange Protocol (DLEP):** Recent work has focused on aligning the CONFINE DLEP implementation to the standardisation work at the IETF and its integration with Community-Lab.

- **Confine monitoring system:** The Confine monitoring system was consolidated in the fourth year with maintenance, stabilization of its functionality and documentation.

All these contributions are deployed and in production use as part of the Community-Lab testbeds, by the community networks involved, and as tools used by experimenters in the area of community networking from academia, industry, community network participants and software developers. The availability of these services will extend during the sustainability period of Community-Lab of one year after the end of the CONFINE project, supported in a voluntary manner.

The developed software can be found in the project repository at [http://redmine.confine-project.eu](http://redmine.confine-project.eu).
Acronyms

BGP Border Gateway Protocol
CN Community Network
CNDB Common Node DataBase
COTS Commercial off-the-shelf
IP Internet Protocol
LXC LinuX Container
UCI Unified Configuration Interface
UPC Universitat Politècnica de Catalunya
WNIC Wireless Network Interface Card
VCT Virtual Confine Testbed
DLEP Dynamic Link Exchange Protocol
API Application Programming Interface
FOSS Free and Open Source Software
JSON JavaScript Object Notation
KVM Kernel-based Virtual Machine
NREN National Research and Education Network
RD Research Device
REST Representational State Transfer
SFA Slice-based Federation Architecture
VLAN Virtual Local Area Network
WiFi Also Wi-Fi, related to wireless networking technology, from the slogan: Wireless Fidelity
ns-3 Network Simulator 3
Bibliography


